Designing an Algorithm for Role Analysis

by

Viktor Kuncak

B.S., University of Novi Sad (2000)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2001

Author . . . .
Department of Electrical Engineering and Computer Science
August 20, 2001

Certified by . . . .
Martin C. Rinard
Associate Professor
Thesis Supervisor

Accepted by . . . .
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Designing an Algorithm for Role Analysis

by
Viktor Kuncak

Submitted to the Department of Electrical Engineering and Computer Science
on August 20, 2001, in partial fulfillment of the
requirements for the degree of
Master of Science

Abstract

This thesis presents a system for specifying constraints on dynamically changing referencing relationships of heap objects, and an analysis for static verification of these constraints. The constraint specification system is based on the concept of role. The role of an object depends, in large part, on its aliasing relationships with other objects, with the role of each object changing as its aliasing relationships change. In this way roles capture object and data structure properties such as unique references, membership of objects in data structures, disjointness of data structures, absence of representation exposure, bidirectional associations, treeness, and absence or presence of cycles in the heap.

Roles generalize linear types by allowing multiple aliases of heap objects that participate in recursive data structures. Unlike graph grammars and graph types, roles contain sufficiently general constraints to conservatively approximate any data structure.

We give a semantics for mutually recursive role definitions and derive properties of roles as an invariant specification language. We introduce a programming model that allows temporary violations of role constraints. We describe a static role analysis for verifying that a program conforms to the programming model. The analysis uses fixpoint computation to synthesize loop invariants in each procedure.

We introduce a procedure interface specification language and its semantics. We present an interprocedural, compositional, and context-sensitive role analysis that verifies that a program respects the role constraints across procedure calls.

Thesis Supervisor: Martin C. Rinard
Title: Associate Professor

Acknowledgments

I would like to thank my advisor Martin Rinard for initiating this exciting research direction. He proposed the concept of roles and has been providing the essential guidance in exploring the large design space of lightweight program specifications.

I am grateful to all members of Martin's group for creating a stimulating atmosphere for research. I thank Patrick Lam who dared to join me in the exploration of roles while the concept was still in its early stage. He helped crystallize ideas of the role analysis and provided a valuable help in its presentation. I thank Brian Demsky for interesting discussions related to dynamic role discovery. I thank Darko Marinov for valuable critical comments regarding the concept of roles and role analysis. I thank Alexandru Sălcianu for discussions about the interprocedural aspects of the pointer and escape analysis. I thank Radu Rugină for answering my questions about the context sensitive pointer analyses, C. Scott Ananian for thoughts on introducing sets and relations into Java, Chandrasekhar Boyapati for his comments on relationships of roles with type systems, and William Beebee for discussions about region-based memory allocation in Java.

Many thanks to Darko, Alexandru, and Maria-Cristina Marinescu for helping me survive in the new environment. I also thank my office mates Karen Zee, Patrick Lam and Jonathan Babb for tolerating my continuous presence in the office. I thank Jon for many intriguing chats on a range of topics.

I thank my parents and family for all the support they gave me over the previous years and this year in particular.

Contents

List of Figures

Chapter 1

Introduction

Types capture important properties of the objects that programs manipulate, increasing both the safety and readability of the program. Traditional type systems capture properties (such as the format of data items stored in the fields of the object) that are invariant over the lifetime of the object. But in many cases, properties that do change are as important as properties that do not. Recognizing the benefit of capturing these changes, researchers have developed systems in which the type of the object changes as the values stored in its fields change or as the program invokes operations on the object [84, 83, 20, 91, 92, 11, 40, 26]. These systems integrate the concept of changing object states into the type system.

The fundamental idea in this work is that the state of each object also depends on the data structures in which it participates. Our type system therefore captures the referencing relationships that determine this data structure participation. As objects move between data structures, their types change to reflect their changing relationships with other objects. Our system uses *roles* to formalize the concept of a type that depends on the referencing relationships. Each role declaration provides complete aliasing information for each object that plays that role—in addition to specifying roles for the fields of the object, the role declaration also identifies the complete set of references in the heap that refer to the object. In this way roles generalize linear type systems [87, 6, 56] by allowing multiple aliases to be statically tracked, and extend alias types [82, 88] with the ability to specify roles of objects that are the source of aliases.

This approach attacks a key difficulty associated with state-based type systems: the need to ensure that any state change performed using one alias is correctly reflected in the declared types of the other aliases. Because each object's role identifies all of its heap aliases, the analysis can verify the correctness of the role information at all remaining or new heap aliases after an operation changes the referencing relationships.

Roles capture important object and data structure properties, improving both the safety and transparency of the program. For example, roles allow the programmer to express data structure consistency properties (with the properties verified by the role analysis), to improve the precision of procedure interface specifications (by allowing the programmer to specify the role of each parameter), to express precise referenc-

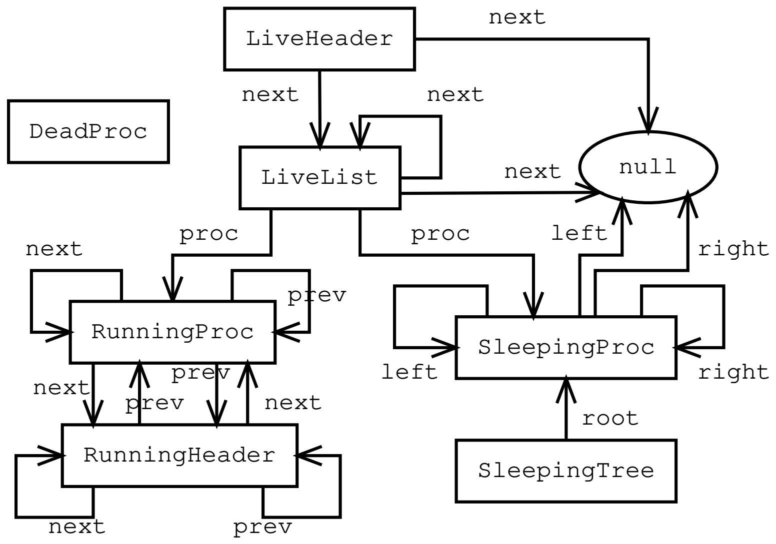


Figure 1-1: Role Reference Diagram for a Scheduler

ing and interaction behaviors between objects (by specifying verified roles for object fields and aliases), and to express constraints on the coordinated movements of objects between data structures (by using the aliasing information in role definitions to identify legal data structure membership combinations). Roles may also aid program optimization by providing precise aliasing information.

1.1 Overview of Roles

Figure 1-1 presents a *role reference diagram* for a process scheduler. Each box in the diagram denotes a disjoint set of objects of a given role. The labelled arrows between boxes indicate possible references between the objects in each set. As the diagram indicates, the scheduler maintains a list of live processes. A live process can be either running or sleeping. The running processes form a doubly-linked list, while sleeping processes form a binary tree. Both kinds of processes have `proc` references from the live list nodes `LiveList`. Header objects `RunningHeader` and `SleepingTree` simplify operations on the data structures that store the process objects.

As Figure 1-1 shows, data structure participation determines the conceptual state of each object. In our example, processes that participate in the sleeping process tree data structure are classified as sleeping processes, while processes that participate in the running process list data structure are classified as running processes. Moreover, movements between data structures correspond to conceptual state changes—when a process stops sleeping and starts running, it moves from the sleeping process tree to the running process list.

1.1.1 Role Definitions

Figure 1-2 presents the role definitions for the objects in our example.[1] Each role definition specifies the constraints that an object must satisfy to play the role. Field constraints specify the roles of the objects to which the fields refer, while slot constraints identify the number and kind of aliases of the object.

Role definitions may also contain two additional kinds of constraints: identity constraints, which specify paths that lead back to the object, and acyclicity constraints, which specify paths with no cycles. In our example, the identity constraint `next.prev` in the `RunningProc` role specifies the cyclic doubly-linked list constraint that following the `next`, then `prev` fields always leads back to the initial object. The acyclic constraint `left, right` in the `SleepingProc` role specifies that there are no cycles in the heap involving only `left` and `right` edges. On the other hand, the list of running processes must be cyclic because its nodes can never point to `null`.

The slot constraints specify the complete set of heap aliases for the object. In our example, this implies that no process can be simultaneously running and sleeping.

In general, roles can capture data structure consistency properties such as disjointness and can prevent representation exposure [14, 21]. As a data structure description language, roles can naturally specify trees with additional pointers. Roles can also approximate non-tree data structures like sparse matrices. Because most role constraints are local, it is possible to inductively infer them from data structure instances.

1.1.2 Roles and Procedure Interfaces

Procedures specify the initial and final roles of their parameters. The `suspend` procedure in Figure 1-3, for example, takes two parameters: an object with role `RunningProc p`, and the `SleepingTree s`. The procedure changes the role of the object referenced by `p` to `SleepingProc` whereas the object referenced by `s` retains its original role. To perform the role change, the procedure removes `p` from its `RunningList` data structure and inserts it into the `SleepingTree` data structure `s`. If the procedure fails to perform the insertions or deletions correctly, for instance by leaving an object in both structures, the role analysis will report an error.

1.2 Contributions

This thesis makes the following contributions:

- **Role Concept:** The concept that the state of an object depends on its referencing relationships; specifically, that objects with different heap aliases should be regarded as having different states.

[1] In general, each role definition would specify the static class of objects that can play that role. To simplify the presentation, we assume that all objects are instances of a single class with a set of fields F.

```
role LiveHeader {
  fields next : LiveList | null;
}
role LiveList {
  fields next : LiveList | null,
         proc : RunningProc | SleepingProc;
  slots  LiveList.next | LiveHeader.next;
  acyclic next;
}
role RunningHeader {
  fields next : RunningProc | RunningHeader,
         prev : RunningProc | RunningHeader;
  slots  RunningHeader.next | RunningProc.next,
         RunningHeader.prev | RunningProc.prev;
  identities next.prev, prev.next;
}
role RunningProc {
  fields next : RunningProc | RunningHeader,
         prev : RunningProc | RunningHeader;
  slots  RunningHeader.next | RunningProc.next,
         RunningHeader.prev | RunningProc.prev,
         LiveList.proc;
  identities next.prev, prev.next;
}
role SleepingTree {
  fields root : SleepingProc | null,
  acyclic left, right;
}
role SleepingProc {
  fields left  : SleepingProc | null,
         right : SleepingProc | null;
  slots SleepingProc.left | SleepingProc.right |
        SleepingTree.root;
        LiveList.proc;
  acyclic left, right;
}
role DeadProc {  }
```

Figure 1-2: Role Definitions for a Scheduler

```
procedure suspend(p : RunningProc ->> SleepingProc,
                  s : SleepingTree)
local pp, pn, r;
{
   pp = p.prev;   pn = p.next;
   r = s.root;
   p.prev = null; p.next = null;
   pp.next = pn;  pn.prev = pp;
   s.root = p;    p.left = r;
   setRole(p : SleepingProc);
}
```

Figure 1-3: Suspend Procedure

- **Role Semantics and its Consequences:** It presents a semantics of a language for defining roles. The programmer can use this language to express data structure invariants and properties such as participation of objects in data structures. We show how roles can be used to control the aliasing of objects, and express reachability properties. We show certain decidability and undecidability results for roles.
- **Programming Model:** It presents a set of role consistency rules. These rules give a programming model for changing the role of an object and the circumstances under which roles can be temporarily violated.
- **Procedure Interface Specification Language:** It presents a language for specifying the initial context and effects of each procedure. The effects summarize the actions of the procedure in terms of the references it changes and the regions of the heap that it affects.
- **Role Analysis Algorithm:** It presents an algorithm for verifying that the program respects the constraints given by a set of role definitions and procedure specifications. The algorithm uses a data-flow analysis to infer intermediate referencing relationships between objects, allowing the programmer to focus on role changes and procedure interfaces. The analysis can verify acyclicity constraints even if they are temporarily violated. The interprocedural analysis verifies read effects as well as "may" and "must" write effects by maintaining a fine grained mapping between the current heap and the initial context of the procedure.

1.3 Outline of the Thesis

The rest of the thesis is organized as follows.

In Chapter 2 we introduce the representation of program heap (2.1.1) and the representation of role constraints introduced by the role definitions (2.1.2). We formally define the semantics of roles by giving a criterion for a heap to satisfy the role constraints (2.1.3). We then highlight some application level properties that can be specified using roles (2.2) and give examples of using roles to describe data structures. We give a list of properties (2.3) that show how roles help control aliasing while giving more flexibility than linear type systems. We show how to deduce reachability properties from role constraints and give a criterion for a set of roles to define a tree. A more detailed study of the constraints expressible using roles is delegated to Appendix A, where we prove decidability of the satisfiability problem for a class of role constraints (A.1.4), and undecidability of the model inclusion for role definitions (A.2).

In Chapter 3 we introduce a programming model that enables role definitions to be integrated with the program. We introduce a core programming language with procedures (3.1) and give its operational semantics (3.2). Next we introduce the notion of onstage and offstage nodes (3.3) which defines the criterion for temporary violations of role constraints by generalizing heap consistency from (2.1.3). As part of the programming model we introduce restrictions on programs that simplify later analysis and ensure role consistency across procedure calls (3.4). We give the preconditions for transitions of the operational semantics that formalize role consistency. We then introduce an instrumented semantics that gives the programmer complete control over the assignment of roles to objects (3.5). This completes the description of the programming model, which is verified by the role analysis.

We present the intraprocedural role analysis in Chapter 4. We define the abstract representation of concrete heaps called role graphs and specify the abstraction relation (4.1). We then define transfer functions for the role analysis (4.2). This includes the expansion relation (4.2.1) used to instantiate nodes from offstage to onstage using instantiation (4.2.1) and split (4.2.1). We model the movement of nodes offstage using the contraction relation (4.2.2). We also describe the checks that the role analysis performs on role graphs to ensure that the program respects the programming model (4.2.3, 4.2.4).

In Chapter 5 we generalize the role analysis to the interprocedural case. We first introduce procedure interface specification language (5.1) that describes initial context (5.1.1) and effects (5.1.2) of each procedure. We give examples of procedure interfaces and define the semantics of initial contexts (5.1.1) and effects (5.1.3). The interprocedural analysis extends the intraprocedural analysis from Chapter 4 by verifying that each procedure respects its specification (5.2) and by instantiating procedure specifications to analyze call sites (5.3). The verification of transfer relations uses a fine grained mapping between nodes of the role graph at each program point and nodes of the initial context. The analysis of call sites needs to establish the mapping between the current role graphs and callee's initial context (5.3.1), instantiate callee's effects (5.3.2) and then reconstruct the roles of modified non-parameter nodes (5.3.3).

In Chapter 6 we present the extensions of the basic role framework described in previous chapters. These extensions allow a statically unbounded number of heap references to objects (6.1), roles defined by references from local variables, non-incremental changes to the role assignment (6.4), and roles for specifying partial information about object's fields and aliases (6.5). The last section also outlines a subtyping criterion for partial roles.

In Chapter 7 we compare our work to the previous typestate systems, the proposals to control the aliasing in object oriented programming and the term roles as used in object modeling and database community. We compare our role analysis with program verification and analysis techniques for dynamically allocated data structures. Chapter 8 concludes the thesis.

Chapter 2

Roles as a Constraint Specification Language

In this chapter we introduce the formal semantics of roles. We then show how to use roles to specify properties of objects and data structures.

2.1 Abstract Syntax and Semantics of Roles

In this section, we precisely define what it means for a given heap to satisfy a set of role definitions. In subsequent sections we will use this definition as a starting point for a programming model and role analysis.

2.1.1 Heap Representation

We represent a concrete program heap as a finite directed graph H_c with $\mathsf{nodes}(H_c)$ representing objects of the heap and labelled edges representing heap references. A graph edge $\langle o_1, f, o_2 \rangle \in H_c$ denotes a reference with field name f from object o_1 to object o_2. To simplify the presentation, we fix a global set of fields F and assume that all objects have the set of fields F.

2.1.2 Role Representation

Let R denote the set of roles used in role definitions, null_R be a special symbol always denoting a null object null_c, and let $R_0 = R \cup \{\mathsf{null}_R\}$. We represent each role as the conjunction of the following four kinds of constraints:

- **Fields:** For every field name $f \in F$ we introduce a function $\mathsf{field}_f : R \to 2^{R_0}$ denoting the set of roles that objects of role $r \in R$ can reference through field f. A field f of role r can be null if and only if $\mathsf{null}_R \in \mathsf{field}_f(r)$. The explicit use of null_R and the possibility to specify a set of alternative roles for every field allows roles to express both may and must referencing relationships.

- **Slots:** Every role r has $\mathsf{slotno}(r)$ slots. A slot $\mathsf{slot}_k(r)$ of role $r \in R$ is a subset of $R \times F$. Let o be an object of role r and o' an object of role r'. A reference $\langle o', f, o \rangle \in H_c$ can fill a slot k of object o if and only if $\langle r', f \rangle \in \mathsf{slot}_k(r)$. An object with role r must have each of its slots filled by exactly one reference.

- **Identities:** Every role $r \in R$ has a set of $\mathsf{identities}(r) \subseteq F \times F$. Identities are pairs of fields $\langle f, g \rangle$ such that following reference f on object o and then returning on reference g leads back to o.

- **Acyclicities:** Every role $r \in R$ has a set $\mathsf{acyclic}(r) \subseteq F$ of fields along which cycles are forbidden.

2.1.3 Role Semantics

We define the semantics of roles as a conjunction of invariants associated with role definitions. A *concrete role assignment* is a map $\rho_c : \mathsf{nodes}(H_c) \to R_0$ such that $\rho_c(\mathsf{null}_c) = \mathsf{null}_R$.

Definition 1 *Given a set of role definitions, we say that heap H_c is* role consistent *iff there exists a role assignment $\rho_c : \mathsf{nodes}(H_c) \to R_0$ such that for every $o \in \mathsf{nodes}(H_c)$ the predicate $\mathsf{locallyConsistent}(o, H_c, \rho_c)$ is satisfied. We call any such role assignment ρ_c a* valid *role assignment.*

The predicate $\mathsf{locallyConsistent}(o, H_c, \rho_c)$ formalizes the constraints associated with role definitions.

Definition 2 $\mathsf{locallyConsistent}(o, H_c, \rho_c)$ *iff all of the following conditions are met. Let $r = \rho_c(o)$.*

1) *For every field $f \in F$ and $\langle o, f, o' \rangle \in H_c$, $\rho_c(o') \in \mathsf{field}_f(r)$.*
2) *Let $\{\langle o_1, f_1 \rangle, \ldots, \langle o_k, f_k \rangle\} = \{\langle o', f \rangle \mid \langle o', f, o \rangle \in H_c\}$ be the set of all aliases of node o. Then $k = \mathsf{slotno}(r)$ and there exists some permutation p of the set $\{1, \ldots, k\}$ such that $\langle \rho_c(o_i), f_i \rangle \in \mathsf{slot}_{p_i}(r)$ for all i.*
3) *If $\langle o, f, o' \rangle \in H_c$, $\langle o', g, o'' \rangle \in H_c$, and $\langle f, g \rangle \in \mathsf{identities}(r)$, then $o = o''$.*
4) *It is not the case that graph H_c contains a cycle $o_1, f_1, \ldots, o_s, f_s, o_1$ where $o_1 = o$ and $f_1, \ldots, f_s \in \mathsf{acyclic}(r)$*

Note that a role consistent heap may have multiple valid role assignments ρ_c. However, in each of these role assignments, every object o is assigned exactly one role $\rho_c(o)$. The existence of a role assignment ρ_c with the property $\rho_c(o_1) \neq \rho_c(o_2)$ thus implies $o_1 \neq o_2$. This is just one of the ways in which roles make aliasing more predictable.

2.2 Using Roles

Roles capture important properties of the objects and provide useful information about how the actions of the program affect those properties.

- **Consistency Properties:** Roles can ensure that the program respects application - level data structure consistency properties. The roles in our process scheduler, for example, ensure that a process cannot be simultaneously sleeping and running.

- **Interface Changes:** In many cases, the interface of an object changes as its referencing relationships change. In our process scheduler, for example, only running processes can be suspended. Because procedures declare the roles of their parameters, the role system can ensure that the program uses objects correctly even as the object's interface changes.

- **Multiple Uses:** Code factoring minimizes code duplication by producing general-purpose classes (such as the Java Vector and Hashtable classes) that can be used in a variety of contexts. But this practice obscures the different purposes that different instances of these classes serve in the computation. Because each instance's purpose is usually reflected in its relationships with other objects, roles can often recapture these distinctions.

- **Correlated Relationships:** In many cases, groups of objects cooperate to implement a piece of functionality. Standard type declarations provide some information about these collaborations by identifying the points-to relationships between related objects at the granularity of classes. But roles can capture a much more precise notion of cooperation, because they track correlated state changes of related objects.

Programmers can use roles for specifying the membership of objects in data structures and the structural invariants of data structures. In both cases, the slot constraints are essential.

When used to describe membership of an object in a data structure, slots specify the source of the alias from a data structure node that stores the object. By assigning different sets of roles to data structures used at different program points, it is possible to distinguish nodes stored in different data structure instances. As an object moves between data structures, the role of the object changes appropriately to reflect the new source of the alias.

When describing nodes of data structures, slot constraints specify the aliasing constraints of nodes; this is enough to precisely describe a variety of data structures and approximate many others. Property 16 below shows how to identify trees in role definitions even if tree nodes have additional aliases from other sets of nodes. It is also possible to define nodes which make up a compound data structure linked via disjoint sets of fields, such as threaded trees, sparse matrices and skip lists.

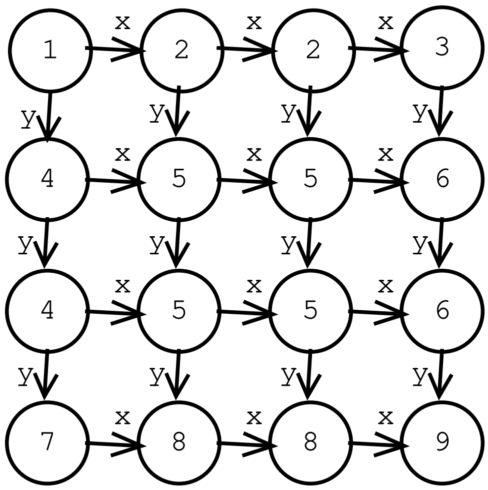


Figure 2-1: Roles of Nodes of a Sparse Matrix

Example 3 The following role definitions specify a sparse matrix of width and height at least 3. These definitions can be easily constructed from a sketch of a sparse matrix in Figure 2-1.

```
role A1 {
  fields x : A2, y : A4;
  acyclic x, y;
}
role A2 {
  fields x : A2 | A3, y : A5;
  slots A1.x | A2.x;
  acyclic x, y;
}
role A3 {
  fields y : A6;
  slots A2.x;
  acyclic x, y;
}
role A4 {
  fields x : A5, y : A4 | A7;
  slots A1.y | A4.y;
  acyclic x, y;
}
role A5 {
  fields x : A5 | A6, y : A5 | A8;
  slots A4.x | A5.x, A2.y | A5.y;
  acyclic x, y;
```

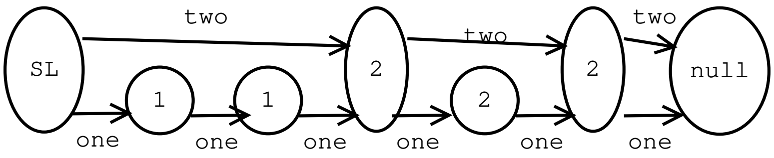


Figure 2-2: Sketch of a Two-Level Skip List

```
}
role A6 {
  fields y : A6 | A9;
  slots A5.x, A3.y | A6.y;
  acyclic x, y;
}
role A7 {
  fields x : A8;
  slots A4.y;
  acyclic x, y;
}
role A8 {
  fields x : A8 | A9;
  slots A7.x | A8.x, A5.y;
  acyclic x, y;
}
role A9 {
  slots A8.x, A6.y;
  acyclic x, y;
}
```

△

Example 4 We next give role definitions for a two-level skip list [69] sketched in Figure 2-2.

```
role SkipList {
  fields one : OneNode | TwoNode | null;
         two : TwoNode | null;
}
role OneNode {
  fields one : OneNode | TwoNode | null;
         two : null;
  slots OneNode.one | TwoNode.one | SkipList.one;
  acyclic one, two;
}
```

```
role TwoNode {
  fields one : OneNode | TwoNode | null;
         two : TwoNode | null;
  slots OneNode.one | TwoNode.one | SkipList.one,
        TwoNode.two | SkipList.two;
  acyclic one, two;
}
```

$\triangle$

2.3 Some Simple Properties of Roles

In this section we identify some of the invariants expressible using sets of mutually recursive role definitions. Some further properties of roles are given in Appendix A.

The following properties show some of the ways role specifications make object aliasing more predictable. They are an immediate consequence of the semantics of roles.

Property 5 *(Role Disjointness)*
If there exists a valid role assignment ρ_c for H_c such that $\rho(o_1) \neq \rho(o_2)$, then $o_1 \neq o_2$.

The previous property gives a simple criterion for showing that objects o_1 and o_2 are unaliased: find a valid role assignment which assigns different roles to o_1 and o_2. This use of roles generalizes the use of static types for pointer analysis [24]. Since roles create a finer partition of objects than a typical static type system, their potential for proving absence of aliasing is even larger.

Property 6 *(Disjointness Propagation)*
If $\langle o_1, f, o_2\rangle, \langle o_3, g, o_4\rangle \in H_c$, $o_1 \neq o_3$, and there exists a valid role assignment ρ_c for H_c such that $\rho_c(o_2) = \rho_c(o_4) = r$ but $\mathsf{field}_f(r) \cap \mathsf{field}_g(r) = \emptyset$, then $o_2 \neq o_4$.

Property 7 *(Generalized Uniqueness)*
If $\langle o_1, f, o_2\rangle, \langle o_3, g, o_4\rangle \in H_c$, $o_1 \neq o_3$, and there exists a role assignment ρ_c such that $\rho_c(o_2) = \rho_c(o_4) = r$, but there are no indices $i \neq j$ such that $\langle \rho_c(o_1), f\rangle \in \mathsf{slot}_i(r)$ and $\langle \rho_c(o_2), g\rangle \in \mathsf{slot}_j(r)$ then $o_2 \neq o_4$.

A special case of Property 7 occurs when $\mathsf{slotno}(r) = 1$; this constrains all references to objects of role r to be unique.

Role definitions induce a role reference diagram RRD which captures some, but not all, role constraints.

Definition 8 *(Role Reference Diagram)*
Given a set of definitions of roles R, a role reference diagram RRD *is is a directed graph with nodes R_0 and labelled edges defined by*

$$\begin{aligned} \mathsf{RRD} = & \{\langle r, f, r'\rangle \mid r' \in \mathsf{field}_f(r) \textit{ and } \exists i \ \langle r, f\rangle \in \mathsf{slot}_i(r')\} \\ & \cup \{\langle r, f, \mathsf{null}_R\rangle \mid \mathsf{null}_R \in \mathsf{field}_f(r)\} \end{aligned}$$

Each role reference diagram is a refinement of the corresponding class diagram in a statically typed language, because it partitions classes into multiple roles according to their referencing relationships. The sets $\rho_c^{-1}(r)$ of objects with role r change during program execution, reflecting the changing referencing relationships of objects.

Role definitions give more information than a role reference diagram. Slot constraints specify not only that objects of role r_1 can reference objects of role r_2 along field f, but also give cardinalities on the number of references from other objects. In addition, role definitions include identity and acyclicity constraints, which are not present in role reference diagrams.

Property 9 *Let ρ_c be any valid role assignment. Define*

$$G = \{\langle \rho_c(o_1), f, \rho_c(o_2)\rangle \mid \langle o_1, f, o_2\rangle \in H_c\}$$

Then G is a subgraph of RRD.

It follows from Property 9 that roles give an approximation of may-reachability among heap objects.

Property 10 *(May Reachability)*
If there is a valid role assignment $\rho_c : \mathsf{nodes}(H_c) \to R_0$ such that $\rho_c(o_1) \neq \rho_c(o_2)$ where $o_1, o_2 \in \mathsf{nodes}(H_c)$ and there is no path from $\rho_c(o_1)$ to $\rho_c(o_2)$ in the role reference diagram RRD, then there is no path from o_1 to o_2 in H_c.

The next property shows the advantage of explicitly specifying null references in role definitions. While the ability to specify acyclicity is provided by the `acyclic` constraint, it is also possible to indirectly specify must-cyclicity.

Property 11 *(Must Cyclicity)*
Let $F_0 \subseteq F$ and $R_{\mathsf{CYC}} \subseteq R$ be a set of nodes in the role reference diagram RRD such that for every node $r \in R_{\mathsf{CYC}}$, if $\langle r, f, r'\rangle \in \mathsf{RRD}$ then $r' \in R_{\mathsf{CYC}}$. If ρ_c is a valid role assignment for H_c, then every object $o_1 \in H_c$ with $\rho_c(o_1) \in R_{\mathsf{CYC}}$ is a member of a cycle in H_c with edges from F_0.

The following property shows that roles can specify a form of must-reachability among the sets of objects with the same role.

Property 12 *(Downstream Path Termination)*
Assume that for some set of fields $F_0 \subseteq F$ there are sets of nodes $R_{\mathsf{INTER}} \subseteq R$, $R_{\mathsf{FINAL}} \subseteq R_0$ of the role reference diagram RRD such that for every node $r \in R_{\mathsf{INTER}}$:

1. $F_0 \subseteq \mathsf{acyclic}(r)$
2. *if $\langle r, f, r'\rangle \in \mathsf{RRD}$ for $f \in F_0$, then $r' \in R_{\mathsf{INTER}} \cup R_{\mathsf{FINAL}}$*

Let ρ_c be a valid role assignment for H_c. Then every path in H_c starting from an object o_1 with role $\rho_c(o_1) \in R_{\mathsf{INTER}}$ and containing only edges labelled with F_0 is a prefix of a path that terminates at some object o_2 with $\rho_c(o_2) \in R_{\mathsf{FINAL}}$.

Property 13 *(Upstream Path Termination)*
Assume that for some set of fields $F_0 \subseteq F$ there are sets of nodes $R_{\mathsf{INTER}} \subseteq R$, $R_{\mathsf{INIT}} \subseteq R_0$ of the role reference diagram RRD *such that for every node $r \in R_{\mathsf{INTER}}$:*

1. $F_0 \subseteq \mathsf{acyclic}(r)$
2. *if $\langle r', f, r\rangle \in \mathsf{RRD}$ for $f \in F_0$, then $r' \in R_{\mathsf{INTER}} \cup R_{\mathsf{INIT}}$*

Let ρ_c be a valid role assignment for H_c. Then every path in H_c terminating at an object o_2 with $\rho_c(o_2) \in R_{\mathsf{INTER}}$ and containing only edges labelled with F_0 is a suffix of a path which started at some object o_1, where $\rho_c(o_1) \in R_{\mathsf{INIT}}$.

We next describe the conditions that guarantee the existence at least one path in the heap, rather than stating the properties of all paths as in Properties 12 and 13.

Property 14 *(Downstream Must Reachability)*
Assume that for some set of fields $F_0 \subseteq F$ there are sets of roles $R_{\mathsf{INTER}} \subseteq R$, $R_{\mathsf{FINAL}} \subset R_0$ of the role reference diagram RRD *such that for every node $r \in R_{\mathsf{INTER}}$:*

1. $F_0 \subseteq \mathsf{acyclic}(r)$
2. *there exists $f \in F_0$ such that $\mathsf{field}_f(r) \subseteq R_{\mathsf{INTER}} \cup R_{\mathsf{FINAL}}$*

Let ρ_c be a valid role assignment for H_c. Then for every object o_1 with $\rho_c(o_1) \in R_{\mathsf{INTER}}$ there is a path in H_c with edges from F_0 from o_1 to some object o_2 where $\rho_c(o_2) \in R_{\mathsf{FINAL}}$.

Property 15 *(Upstream Must Reachability)*
Assume that for some set of fields $F_0 \subseteq F$ there are sets of nodes $R_{\mathsf{INTER}} \subseteq R$, $R_{\mathsf{INIT}} \subseteq R$ of the role reference diagram RRD *such that for every node $r \in R_{\mathsf{INTER}}$:*

1. $F_0 \subseteq \mathsf{acyclic}(r)$
2. *there exists k such that $\mathsf{slot}_k(r) \subseteq (R_{\mathsf{INTER}} \cup R_{\mathsf{INIT}}) \times F$*

Let ρ_c be a valid role assignment for H_c. Then for every object o_2 with $\rho_c(o_2) \in R_{\mathsf{INTER}}$ there is a path in H_c from some object o_1 with $\rho_c(o_1) \in R_{\mathsf{INIT}}$ to the object o_2.

Trees are a class of data structures especially suited for static analysis. Roles can express graphs that are not trees, but it is useful to identify trees as certain sets of mutually recursive role definitions.

Property 16 *(Treeness)*
Let $R_{\mathsf{TREE}} \subseteq R$ be a set of roles and $F_0 \subseteq F$ set of fields such that for every $r \in R_{\mathsf{TREE}}$

1. $F_0 \subseteq \mathsf{acyclic}(r)$
2. $|\{i \mid \mathsf{slot}_i(r) \cap (R_{\mathsf{TREE}} \times F_0) \neq \emptyset\}| \leq 1$

Let ρ_c be a valid role assignment for H_c and

$$S \subseteq \{\langle n_1, f, n_2\rangle \mid \langle n_1, f, n_2\rangle \in H_c, \rho(n_1), \rho(n_2) \in R_{\mathsf{TREE}}, f \in F_0\}$$

Then S is a set of trees.

Chapter 3

A Programming Model

In this chapter we define what it means for an execution of a program to respect the role constraints. This definition is complicated by the need to allow the program to temporarily violate the role constraints during data structure manipulations. Our approach is to let the program violate the constraints for objects referenced by local variables or parameters, but require all other objects to satisfy the constraints.

We first present a simple imperative language with dynamic object allocation and give its operational semantics. We then specify additional statement preconditions that enforce the role consistency requirements.

3.1 A Simple Imperative Language

Our core language contains, as basic statements, Load (`x=y.f`), Store (`x.f=y`), Copy (`x=y`), and New (`x=new`). All variables are references to objects in the global heap and all assignments are reference assignments. We use an elementary `test` statement combined with nondeterministic choice and iteration to express `if` and `while` statement, using the usual translation [44, 5] given in Figure 3-1. We represent the control flow of programs using control-flow graphs.

A program is a collection of procedures $\mathsf{proc} \in \mathsf{Proc}$. Procedures change the global heap but do not return values. Every procedure proc has a list of parameters $\mathsf{param}(\mathsf{proc}) = \{\mathsf{param}_i(\mathsf{proc})\}_i$ and a list of local variables $\mathsf{local}(\mathsf{proc})$. We use $\mathsf{var}(\mathsf{proc})$ to denote $\mathsf{param}(\mathsf{proc}) \cup \mathsf{local}(\mathsf{proc})$. A procedure definition specifies the initial role $\mathsf{preR}_k(\mathsf{proc})$ and the final role $\mathsf{postR}_k(\mathsf{proc})$ for every parameter $\mathsf{param}_k(\mathsf{proc})$. We use proc_j for indices $j \in \mathcal{N}$ to denote activation records of procedure proc. We further assume that there are no modifications of parameter variables so every parameter references the same object throughout the lifetime of procedure activation.

$$
\begin{array}{rcl}
\texttt{if}\ \mathsf{t}\ \mathsf{stat}_1\ \mathsf{stat}_2 & \equiv & (\texttt{test(t)};\mathsf{stat}_1)\,|\,(\texttt{test(!t)};\mathsf{stat}_2) \\
\texttt{while}\ \mathsf{t}\ \mathsf{stat} & \equiv & (\texttt{test(t)};\mathsf{stat})*;\texttt{test(!t)}
\end{array}
$$

Figure 3-1: Syntactic Sugar for `if` and `while`

Example 17 The following `kill` procedure removes a process from both the doubly linked list of running processes and the list of all active processes. This is indicated by the transition from `RunningProc` to `DeadProc`.

```
procedure kill(p : RunningProc ->> DeadProc,
               l : LiveHeader)
local prev, current, cp, nxt, lp, ln;
{
  // find 'p' in 'l'
  prev = l; current = l.next;
  cp = current.proc;
  while (cp != p) {
    prev = current;
    current = current.next;
    cp = current.proc;
  }
  // remove 'current' and 'p' from active list
  nxt = current.next;
  prev.next = nxt; current.
  current.proc = null;
  setRole(current : IsolatedCell);
  // remove 'p' from running list
  lp = p.prev;   ln = p.next;
  p.prev = null; p.next = null;
  lp.next = ln;   ln.prev = lp;
  setRole(p : DeadProc);
}
```

$\triangle$

3.2 Operational Semantics

In this section we give the operational semantics for our language. We focus on the first three columns in Figures 3-2 and 3-3; the safety conditions in the fourth column are detailed in Section 3.4.

Figure 3-2 gives the small-step operational semantics for the basic statements. We use $A \uplus B$ to denote the union $A \cup B$ where the sets A and B are disjoint. The program state consists of the stack s and the concrete heap H_c. The stack s is a sequence of pairs $p@\mathsf{proc}_i \in \times(\mathsf{Proc} \times \mathcal{N})$, where $p \in N_{\mathsf{CFG}}(\mathsf{proc})$ is a program point, and $\mathsf{proc}_i \in \mathsf{Proc} \times \mathcal{N}$ is an activation record of procedure proc. Program points $p \in N_{\mathsf{CFG}}(\mathsf{proc})$ are nodes of the control-flow graphs. There is one control-flow graph for every procedure proc. An edge of the control-flow graph $\langle p, p' \rangle \in E_{\mathsf{CFG}}(\mathsf{proc})$ indicates that control may transfer from point p to point p'. We write $p : \mathsf{stat}$ to state that program point p contains a statement stat. The control flow graph of each procedure contains special program points `entry` and `exit` indicating procedure entry

Statement	Transition	Constraints	Role Consistency
p : `x=y.f`	$\langle p @ \mathsf{proc}_i; s, H_c \uplus \{\langle \mathsf{proc}_i, \mathtt{x}, o_x\rangle\}\rangle \rightarrow$ $\langle p' @ \mathsf{proc}_i; s, H'_c\rangle$	$\mathtt{x}, \mathtt{y} \in \mathsf{local}(\mathsf{proc})$, $\langle \mathsf{proc}_i, \mathtt{y}, o_y\rangle, \langle o_y, \mathtt{f}, o_f\rangle \in H_c$, $\langle p, p'\rangle \in E_{\mathsf{CFG}}(\mathsf{proc})$, $H'_c = H_c \uplus \{\mathsf{proc}_i, \mathtt{x}, o_f\}$	$\mathsf{accessible}(o_f, \mathsf{proc}_i, H_c)$, $\mathsf{con}(H'_c, \mathsf{offstage}(H'_c))$
p : `x.f=y`	$\langle p @ \mathsf{proc}_i; s, H_c \uplus \{\langle o_x, f, o_f\rangle\}\rangle \rightarrow$ $\langle p' @ \mathsf{proc}_i; s, H'_c\rangle$	$\mathtt{x}, \mathtt{y} \in \mathsf{local}(\mathsf{proc})$, $\langle \mathsf{proc}_i, \mathtt{x}, o_x\rangle, \langle \mathsf{proc}_i, \mathtt{y}, o_y\rangle \in H_c$, $\langle p, p'\rangle \in E_{\mathsf{CFG}}(\mathsf{proc})$, $H'_c = H_c \uplus \{\langle o_x, f, o_y\rangle\}$	$o_f \in \mathsf{onstage}(H_c, \mathsf{proc}_i)$ $\mathsf{con}(H'_c, \mathsf{offstage}(H'_c))$
p : `x=y`	$\langle p @ \mathsf{proc}_i; s, H_c \uplus \{\langle \mathsf{proc}_i, \mathtt{x}, o_x\rangle\}\rangle \rightarrow$ $\langle p' @ \mathsf{proc}_i; s, H'_c\rangle$	$\mathtt{x} \in \mathsf{local}(\mathsf{proc})$, $\mathtt{y} \in \mathsf{var}(\mathsf{proc})$, $\langle \mathsf{proc}_i, \mathtt{y}, o_y\rangle \in H_c$, $\langle p, p'\rangle \in E_{\mathsf{CFG}}(\mathsf{proc})$, $H'_c = H_c \uplus \{\langle \mathsf{proc}_i, \mathtt{x}, o_y\rangle\}$	$\mathsf{con}(H'_c, \mathsf{offstage}(H'_c))$
p : `x=new`	$\langle p @ \mathsf{proc}_i; s, H_c \uplus \{\langle \mathsf{proc}_i, \mathtt{x}, o_x\rangle\}\rangle \rightarrow$ $\langle p' @ \mathsf{proc}_i; s, H'_c\rangle$	$\mathtt{x} \in \mathsf{local}(\mathsf{proc})$, o_n fresh, $\langle p, p'\rangle \in E_{\mathsf{CFG}}(\mathsf{proc})$, $H'_c = H_c \uplus \{\langle \mathsf{proc}_i, \mathtt{x}, o_n\rangle\} \uplus \mathsf{nulls}$, $\mathsf{nulls} = \{o_n\} \times F \times \{\mathsf{null}\}$	$\mathsf{con}(H'_c, \mathsf{offstage}(H'_c))$
p : `test(c)`	$\langle p @ \mathsf{proc}_i; s, H_c\rangle \rightarrow$ $\langle p' @ \mathsf{proc}_i; s, H_c\rangle$	$\mathsf{satisfied}_c(\mathtt{c}, \mathsf{proc}_i, H_c)$, $\langle p, p'\rangle \in E_{\mathsf{CFG}}(\mathsf{proc})$	$\mathsf{con}(H_c, \mathsf{offstage}(H_c))$

$$\mathsf{satisfied}_c(\mathtt{x==y}, \mathsf{proc}_i, H_c) \text{ iff } \{o \mid \langle \mathsf{proc}_i, \mathtt{x}, o\rangle \in H_c\} = \{o \mid \langle \mathsf{proc}_i, \mathtt{y}, o\rangle \in H_c\}$$

$$\mathsf{satisfied}_c(\mathtt{!(x==y)}, \mathsf{proc}_i, H_c) \text{ iff not } \mathsf{satisfied}_c(\mathtt{x==y}, \mathsf{proc}_i, H_c)$$

$$\begin{aligned} \mathsf{accessible}(o, \mathsf{proc}_i, H_c) \quad := \quad & (\exists p \in \mathsf{param}(\mathsf{proc}) : \langle \mathsf{proc}_i, p, o\rangle \in H_c) \\ & \text{or not } (\exists \mathsf{proc}'_j\ \exists v \in \mathsf{var}(\mathsf{proc}') : \langle \mathsf{proc}'_j, v, o\rangle \in H_c) \end{aligned}$$

Figure 3-2: Semantics of Basic Statements

Statement	Transition	Constraints	Role Consistency
`entry : _`	$\langle p@\mathsf{proc}_i; s, H_c\rangle \rightarrow$ $\langle p'@\mathsf{proc}_i; s, H_c \uplus \mathsf{nulls}\rangle$	$\mathsf{nulls} = \{\langle \mathsf{proc}_i, v, \mathsf{null}_c\rangle \mid$ $v \in \mathsf{local}(\mathsf{proc}),$ $\langle p, p'\rangle \in E_{\mathsf{CFG}}(\mathsf{proc})$	$\mathsf{con}(H_c, \mathsf{offstage}(H_c))$
$p : \mathsf{proc}'(x_k)_k$	$\langle p@\mathsf{proc}_i; s, H_c\rangle \rightarrow$ $\langle \mathtt{entry}@\mathsf{proc}'_j; p'@\mathsf{proc}_i; s, H'_c\rangle$	j fresh in $p@\mathsf{proc}i; s,$ $\langle p, p'\rangle \in E_{\mathsf{CFG}}(\mathsf{proc}),$ $o_k : \langle \mathsf{proc}_i, x_k, o_k\rangle \in H_c,$ $H'_c = H_c \uplus \{\langle \mathsf{proc}'_j, p_k, o_k\rangle\}_k,$ $\forall k\ \ p_k = \mathsf{param}_k(\mathsf{proc}')$	$\mathsf{conW}(\mathsf{ra}, H_c, S),$ $\mathsf{ra} = \{\langle o_k, \mathsf{preR}_k(\mathsf{proc}')\rangle\}_k,$ $S = \mathsf{offstage}(H_c) \cup \{o_k\}_k$
`exit : _`	$\langle p@\mathsf{proc}_i; s, H_c\rangle \rightarrow$ $\langle s, H_c \setminus \mathsf{AF}\rangle$	$\mathsf{AF} = \{\langle \mathsf{proc}_i, v, n\rangle \mid$ $\langle \mathsf{proc}_i, v, n\rangle \in H_c\}$	$\mathsf{conW}(\mathsf{ra}, H_c, S),$ $\mathsf{ra} = \{\langle \mathsf{parnd}_k(\mathsf{proc}_i), \mathsf{postR}_k(\mathsf{proc})\rangle\}_k,$ $S = \mathsf{offstage}(H_c) \cup$ $\{o \mid \langle \mathsf{proc}_i, v, o\rangle \in H_c\}$

$$\mathsf{parnd}_k(\mathsf{proc}_i) = o \text{ where } \langle \mathsf{proc}_i, \mathsf{param}_k(\mathsf{proc}), o\rangle \in H_c$$

Figure 3-3: Semantics of Procedure Call

and exit, with no statements associated with them. We assume that each condition of a `test` statement is of the form `x==y` or `!(x==y)` where `x` and `y` are either variables or a special constant `null` which always points to the null_c object.

The concrete heap is either an error heap error_c or a non-error heap. A non-error heap $H_c \subseteq N \times F \times N \cup ((\mathsf{Proc} \times \mathcal{N}) \times V \times N)$ is a directed graph with labelled edges, where nodes represent objects and procedure activation records, whereas edges represent heap references and local variables. An edge $\langle o_1, f, o_2\rangle \in N \times F \times N$ denotes a reference from object o_1 to object o_2 via field $f \in F$. An edge $\langle \mathsf{proc}_i, \mathtt{x}, o\rangle \in H_c$ means that local variable `x` in activation record proc_i points to object o.

A load statement `x=y.f` makes the variable `x` point to node o_f, which is referenced by the `f` field of object o_y, which is in turn referenced by variable `y`. A store statement `x.f=y` replaces the reference along field `f` in object o_x by a reference to object o_y that is referenced by `y`. The copy statement `x=y` copies a reference to object o_y into variable `x`. The statement `x=new` creates a new object o_n with all fields initially referencing null_c, and makes `x` point to o_n. The statement `test(c)` allows execution to proceed only if condition `c` is satisfied.

Figure 3-3 shows the semantics of procedure calls. Procedure call pushes new activation record onto stack, inserts it into the heap, and initializes the parameters. Procedure entry initializes local variables. Procedure exit removes the activation record from the heap and the stack.

3.3 Onstage and Offstage Objects

At every program point the set $\mathsf{nodes}(H_c)$ of all objects of heap H_c can be partitioned into:

1. **onstage objects** ($\mathsf{onstage}(H_c)$) referenced by a local variable or parameter of some activation frame

$$\begin{aligned}\mathsf{onstage}(H_c, \mathsf{proc}_i) &:= \{o \mid \exists x \in \mathsf{var}(\mathsf{proc}) \\ &\qquad \langle \mathsf{proc}_i, x, o\rangle \in H_c\} \\ \mathsf{onstage}(H_c) &:= \bigcup_{\mathsf{proc}_i} \mathsf{onstage}(H_c, \mathsf{proc}_i)\end{aligned}$$

2. **offstage objects** ($\mathsf{offstage}(H_c)$) unreferenced by local or parameter variables

$$\mathsf{offstage}(H_c) := \mathsf{nodes}(H_c) \setminus \mathsf{onstage}(H_c)$$

Onstage objects need not have correct roles. Offstage objects must have correct roles assuming some role assignment for onstage objects.

Definition 18 *Given a set of role definitions and a set of objects $S_c \subseteq \mathsf{nodes}(S_c)$, we say that heap H_c is* role consistent *for S_c, and we write $\mathsf{con}(H_c, S_c)$, iff there exists a role assignment $\rho_c : \mathsf{nodes}(H_c) \to R_0$ such that the $\mathsf{locallyConsistent}(o, H_c, \rho_c, S_c)$ predicate is satisfied for every object $o \in S_c$.*

We define $\mathsf{locallyConsistent}(o, H_c, \rho_c, S_c)$ to generalize the $\mathsf{locallyConsistent}(o, H_c, \rho_c)$ predicate, weakening the acyclicity condition.

Definition 19 $\mathsf{locallyConsistent}(o, H_c, \rho_c, S_c)$ *holds iff conditions 1), 2), and 3) of Definition 2 are satisfied and the following condition holds:*

4') It is not the case that graph H_c contains a cycle $o_1, f_1, \ldots, o_s, f_s, o_1$ such that $o_1 = o$, $f_1, \ldots, f_s \in \mathsf{acyclic}(r)$, and additionally $o_1, \ldots, o_s \in S_c$.

Here S_c is the set of onstage objects that are not allowed to create a cycle whereas objects in $\mathsf{nodes}(H_c) \setminus S_c$ are exempt from the acyclicity condition. The predicates $\mathsf{locallyConsistent}(o, H_c, \rho_c, S_c)$ and $\mathsf{con}(H_c, S_c)$ are monotonic in S_c, so a larger S_c implies a stronger invariant. For $S_c = \mathsf{nodes}(H_c)$, consistency for S_c is equivalent with heap consistency from Definition 1. Note that the role assignment ρ_c specifies roles even for objects $o \in \mathsf{nodes}(H_c) \setminus S_c$. This is because the role of o may influence the role consistency of objects in S_c which are adjacent to o.

At procedure calls, the role declarations for parameters restrict the set of potential role assignments. We therefore generalize $\mathsf{con}(H_c, S_c)$ to $\mathsf{conW}(\mathsf{ra}, H_c, S_c)$, which restricts the set of role assignments ρ_c considered for heap consistency.

Definition 20 *Given a set of role definitions, a heap H_c, a set $S_c \subseteq \mathsf{nodes}(H_c)$, and a partial role assignment $\mathsf{ra} \subseteq S_c \to R$, we say that the heap H_c is* consistent with ra for S_c, *and write $\mathsf{conW}(\mathsf{ra}, H_c, S_c)$, iff there exists a (total) role assignment $\rho_c : \mathsf{nodes}(H_c) \to R_0$ such that $\mathsf{ra} \subseteq \rho_c$ and for every object $o \in S_c$ the predicate $\mathsf{locallyConsistent}(o, H_c, \rho_c, S_c)$ is satisfied.*

3.4 Role Consistency

We are now able to precisely state the role consistency requirements that must be satisfied for program execution. The role consistency requirements are in the fourth row of Figures 3-2 and 3-3. We assume the operational semantics is extended with transitions leading to a program state with heap error_c whenever role consistency is violated.

3.4.1 Offstage Consistency

At every program point, we require $\mathsf{con}(H_c, \mathsf{offstage}(H_c))$ to be satisfied. This means that offstage objects have correct roles, but onstage objects may have their role temporarily violated.

3.4.2 Reference Removal Consistency

The Store statement `x.f=y` has the following safety precondition. When a reference $\langle o_x, f, o_f \rangle \in H_c$ for $\langle \mathsf{proc}_j, \mathtt{x}, o_x \rangle \in H_c$, and $\langle o_x, \mathtt{f}, o_f \rangle \in H_c$ is removed from the heap, both o_x and o_f must be referenced from the current procedure activation record. It is sufficient to verify this condition for o_f, as o_x is already onstage by definition. The reference removal consistency condition enables the completion of the role change for o_f after the reference $\langle o_x, f, o_f \rangle$ is removed and ensures that heap references are introduced and removed only between onstage objects.

3.4.3 Procedure Call Consistency

Our programming model ensures role consistency across procedure calls using the following protocol.

A procedure call $\mathsf{proc}'(x_1, ..., x_p)$ in Figure 3-3 requires the role consistency precondition $\mathsf{conW}(\mathsf{ra}, H_c, S_c)$, where the partial role assignment ra requires objects o_k, corresponding to parameters x_k, to have roles $\mathsf{preR}_k(\mathsf{proc}')$ expected by the callee, and $S_c = \mathsf{offstage}(H_c) \cup \{o_k\}_k$ for $\langle \mathsf{proc}_j, x_k, o_k \rangle \in H_c$.

To ensure that the callee proc'_j never observes incorrect roles, we impose an *accessibility condition* for the callee's Load statements (see the fourth column of Figure 3-2). The accessibility condition prohibits access to any object o referenced by some local variable of a stack frame other than proc'_j, unless o is referenced by some parameter of proc'_j. Provided that this condition is not violated, the callee proc'_j only accesses objects with correct roles, even though objects that it does not access may have incorrect roles. In Chapter 5 we show how the role analysis statically ensures that the accessibility condition is never violated.

At the procedure exit point (Figure 3-3), we require correct roles for all objects referenced by the current activation frame proc'_j. This implies that heap operations performed by proc'_j preserve heap consistency for all objects accessed by proc'_j.

Statement	Transition	Constraints	Role Consistency
p : roleCheck($x_1, \ldots, x_n$, ra)	$\langle p@\mathsf{proc}_i; s, H_c\rangle \rightarrow$ $\langle p'@\mathsf{proc}_i; s, H_c\rangle$	$\langle p, p'\rangle \in E_{\mathsf{CFG}}$	$\mathsf{conW}(\mathsf{ra}, H_c, S)$, $S = \mathsf{offstage}(H_c) \cup \{o \mid \langle \mathsf{proc}_i, x_k, o\rangle \in H_c\}$

Figure 3-4: Operational Semantics of Explicit Role Check

3.4.4 Explicit Role Check

The programmer can specify a stronger invariant at any program point using statement roleCheck($x_1, \ldots, x_p$, ra). As Figure 3-4 indicates, roleCheck requires the $\mathsf{conW}(\mathsf{ra}, H_c, S_c)$ predicate to be satisfied for the supplied partial role assignment ra where $S_c = \mathsf{offstage}(H_c) \cup \{o_k\}_k$ for objects o_k referenced by given local variables x_k.

3.5 Instrumented Semantics

We expect the programmer to have a specific role assignment in mind when writing the program, with this role assignment changing as the statements of the program change the referencing relationships. So when the programmer wishes to change the role of an object, he or she writes a program that brings the object onstage, changes its referencing relationships so that it plays a new role, then puts it offstage in its new role. The roles of other objects do not change.[1]

To support these programmer expectations, we introduce an augmented programming model in which the role assignment ρ_c is conceptually part of the program's state. The role assignment changes only if the programmer changes it explicitly using the setRole statement. The augmented programming model has an underlying *instrumented semantics* as opposed to the *original semantics*.

Example 21 The original semantics allows asserting different roles at different program points even if the structure of the heap was not changed, as in the following procedure foo.

```
role A1 { fields f : B1; }
role B1 { slots A1.f; }
role A2 { fields f : B2; }
role B2 { slots A2.f; }
procedure foo()
var x, y;
{
  x = new;  y = new;
  x.f = y;
```

[1] An extension to the programming model supports *cascading role changes* in which a single role change propagates through the heap changing the roles of offstage objects, see Section 6.4.

Statement	Transition	Constraints	Role Consistency
p : `x=new`	$\langle p@\mathsf{proc}_i; s, H_c \uplus \{\langle \mathsf{proc}_i, \mathtt{x}, o_x\rangle\}, \rho_c\rangle \rightarrow$ $\langle p'@\mathsf{proc}_i; s, H'_c, \rho'_c\rangle$	$\mathtt{x} \in \mathsf{local}(\mathsf{proc})$, o_n fresh, $\langle p, p'\rangle \in E_{\mathsf{CFG}}(\mathsf{proc})$, $H'_c = H_c$ $\uplus\{\langle \mathsf{proc}_i, \mathtt{x}, o_n\rangle\}$ $\uplus\{o_n\} \times F \times \{\mathsf{null}\}$, $\rho'_c = \rho_c[o_n \mapsto \mathsf{unknown}]$	$\mathsf{conW}(\rho'_c, H'_c, \mathsf{offstage}(H'_c))$
p : `setRole(x:r)`	$\langle p@\mathsf{proc}_i; s, H_c, \rho_c\rangle \rightarrow$ $\langle p'@\mathsf{proc}_i; s, H_c, \rho'_c\rangle$	$\mathtt{x} \in \mathsf{local}(\mathsf{proc}_i)$, $\langle \mathsf{proc}_i, \mathtt{x}, o_x\rangle \in H_c$, $\rho'_c = \rho_c[o_x \mapsto \mathtt{r}]$, $\langle p, p'\rangle \in E_{\mathsf{CFG}}$	$\mathsf{conW}(\rho'_c, H_c, \mathsf{offstage}(H_c))$
p : stat	$\langle s, H_c, \rho_c\rangle \rightarrow$ $\langle s', H'_c, \rho_c\rangle$	$\langle s, H_c\rangle \rightarrow \langle s', H'_c\rangle$	$P \wedge \mathsf{conW}(\rho_c \cup \mathsf{ra}, H''_c, S)$ for every original condition $P \wedge \mathsf{conW}(\mathsf{ra}, H''_c, S)$

Figure 3-5: Instrumented Semantics

```
  roleCheck(x,y, x:A1,y:B1);
  roleCheck(x,y, x:A2,y:B2);
}
```

Both role checks would succeed since each of the specified partial role assignments can be extended to a valid role assignment. On the other hand, the role check statement `roleCheck(x,y, x:A1,y:B2)` would fail.

The procedure `foo` in the instrumented semantics can be written as follows.

```
procedure foo()
var x, y;
{
  x = new;  y = new;
  x.f = y;
  setRole(x:A1);  setRole(y:B1);
  roleCheck(x,y, x:A1,y:B1);
  setRole(x:A2);  setRole(y:B2);
  roleCheck(x,y, x:A2,y:B2);
}
```

The `setRole` statement makes the role change of object explicit. $\triangle$

The instrumented semantics extends the concrete heap H_c with a role assignment ρ_c. Figure 3-5 outlines the changes in instrumented semantics with respect to the original semantics. We introduce a new statement `setRole(x:r)`, which modifies a role assignment ρ_c, giving $\rho_c[o_x \mapsto r]$, where o_x is the object referenced by `x`. All statements other than `setRole` preserve the current role assignment. For every consistency condition $\mathsf{conW}(\mathsf{ra}, H_c, S_c)$ in the original semantics, the instrumented semantics uses the corresponding condition $\mathsf{conW}(\rho_c \cup \mathsf{ra}, H_c, S_c)$ and fails if ρ_c is not an extension of ra. Here we consider $\mathsf{con}(H_c, S)$ to be a shorthand for $\mathsf{conW}(\emptyset, H_c, S)$. For example, the new role consistency condition for the Copy statement $\mathsf{x{=}y}$ is $\mathsf{conW}(\rho_c, H_c, \mathsf{offstage}(H_c))$. The New statement assigns an identifier unknown to the newly created object o_n. By definition, a node with unknown does not satisfy the locallyConsistent predicate. This means that `setRole` must be used to set a a valid role of o_n before o_n moves offstage.

By introducing an instrumented semantics we are not suggesting an implementation that explicitly stores roles of objects at run-time. We instead use the instrumented semantics as the basis of our role analysis and ensure that all role checks can be statically removed. Because the instrumented semantics is more restrictive than the original semantics, our role analysis is a conservative approximation of both the instrumented semantics and the original semantics.

Chapter 4

Intraprocedural Role Analysis

This chapter presents an intraprocedural role analysis algorithm. The goal of the role analysis is to statically verify the role consistency requirements described in the previous chapter.

The key observation behind our analysis algorithm is that we can incrementally verify role consistency of the entire concrete heap H_c by ensuring role consistency for every node when it goes offstage. This allows us to represent the statically unbounded offstage portion of the heap using summary nodes with "may" references. In contrast, we use a "must" interpretation for references from and to onstage nodes. The exact representation of onstage nodes allows the analysis to verify role consistency in the presence of temporary violations of role constraints.

Our analysis representation is a graph in which nodes represent objects and edges represent references between objects. There are two kinds of nodes: *onstage nodes* represent onstage objects, with each onstage node representing one onstage object; and *offstage nodes*, with each offstage node corresponding to a set of objects that play that role. To increase the precision of the analysis, the algorithm occasionally generates multiple offstage nodes that represent disjoint sets of objects playing the same role. Distinct offstage objects with the same role r represent disjoint sets of objects of role r with different reachability properties from onstage nodes.

We frame role analysis as a data-flow analysis operating on a distributive lattice $\mathcal{P}(\mathsf{RoleGraphs})$ of sets of role graphs with set union $\cup$ as the join operator. This chapter focuses on the intraprocedural analysis. We use proc_c to denote the topmost activation record in a concrete heap H_c. In Chapter 5 we generalize the algorithm to the compositional interprocedural analysis.

4.1 Abstraction Relation

Every data-flow fact $\mathcal{G} \subseteq \mathsf{RoleGraphs}$ is a set of role graphs $G \in \mathcal{G}$. Every role graph $G \in \mathsf{RoleGraphs}$ is either a bottom role graph $\bot_G$ representing the set of all concrete heaps (including error_c), or a tuple $G = \langle H, \rho, K \rangle$ representing non-error concrete heaps, where

- $H \subseteq N \times F \times N$ is the abstract heap with nodes N representing objects and fields

F. The abstract heap H represents heap references $\langle n_1, f, n_2 \rangle$ and variables of the currently analyzed procedure $\langle \mathsf{proc}, x, n \rangle$ where $x \in \mathsf{local}(\mathsf{proc})$. Null references are represented as references to abstract node null. We define abstract onstage nodes $\mathsf{onstage}(H) = \{n \mid \langle \mathsf{proc}, x, n \rangle \in H, x \in \mathsf{local}(\mathsf{proc}) \cup \mathsf{param}(\mathsf{proc})\}$ and abstract offstage nodes $\mathsf{offstage}(H) = \mathsf{nodes}(H) \setminus \mathsf{onstage}(H) \setminus \{\mathsf{proc}, \mathsf{null}\}$.

- $\rho : \mathsf{nodes}(H) \to R_0$ is an abstract role assignment, $\rho(\mathsf{null}) = \mathsf{null}_R$;
- $K : \mathsf{nodes}(H) \to \{i, s\}$ indicates the kind of each node; when $K(n) = i$, then n is an individual node representing at most one object, and when $K(n) = s$, n is a summary node representing zero or more objects. We require $K(\mathsf{proc}) = K(\mathsf{null}) = i$, and require all onstage nodes to be individual, $K[\mathsf{onstage}(H)] = \{i\}$.

The abstraction relation α relates a pair $\langle H_c, \rho_c \rangle$ of concrete heap and concrete role assignment with an abstract role graph G.

Definition 22 *We say that an abstract role graph G represents concrete heap H_c with role assignment ρ_c, and write $\langle H_c, \rho_c \rangle \, \alpha \, G$, iff $G = \perp_G$ or: $H_c \neq \mathsf{error}_c$, $G = \langle H, \rho, K \rangle$, and there exists a function $h : \mathsf{nodes}(H_c) \to \mathsf{nodes}(H)$ such that*

1) *H_c is role consistent: $\mathsf{conW}(\rho_c, H_c, \mathsf{offstage}(H_c))$,*
2) *identity relations of onstage nodes with offstage nodes hold: if $\langle o_1, f, o_2 \rangle \in H_c$ and $\langle o_2, g, o_3 \rangle \in H_c$ for $o_1 \in \mathsf{onstage}(H_c)$, $o_2 \in \mathsf{offstage}(H_c)$, and $\langle f, g \rangle \in \mathsf{identities}(\rho_c(o_1))$, then $o_3 = o_1$;*
3) *h is a graph homomorphism: if $\langle o_1, f, o_2 \rangle \in H_c$ then $\langle h(o_1), f, h(o_2) \rangle \in H$;*
4) *an individual node represents at most one concrete object: $K(n) = i$ implies $|h^{-1}(n)| \leq 1$;*
5) *h is bijection on edges which originate or terminate at onstage nodes: if $\langle n_1, f, n_2 \rangle \in H$ and $n_1 \in \mathsf{onstage}(H)$ or $n_2 \in \mathsf{onstage}(H)$, then there exists exactly one $\langle o_1, f, o_2 \rangle \in H_c$ such that $h(o_1) = n_1$ and $h(o_2) = n_2$;*
6) *$h(\mathsf{null}_c) = \mathsf{null}$ and $h(\mathsf{proc}_c) = \mathsf{proc}$;*
7) *the abstract role assignment ρ corresponds to the concrete role assignment: $\rho_c(o) = \rho(h(o))$ for every object $o \in \mathsf{nodes}(H_c)$.*

Note that the error heap error_c can be represented only by the bottom role graph $\perp_G$. The analysis uses $\perp_G$ to indicate a potential role error.

Condition 3) implies that role graph edges are a conservative approximation of concrete heap references. These edges are in general "may" edges. Hence it is possible for an offstage node n that $\langle n, f, n_1 \rangle, \langle n, f, n_2 \rangle \in H$ for $n_1 \neq n_2$. This cannot happen when $n \in \mathsf{onstage}(H)$ because of 5). Another consequence of 5) is that an edge in H from an onstage node n_0 to a summary node n_s implies that n_s represents at least one object. Condition 2) strengthens 1) by requiring certain identity constraints for onstage nodes to hold, as explained in Section 4.2.4.

Example 23 Consider the following role declaration for an acyclic list.

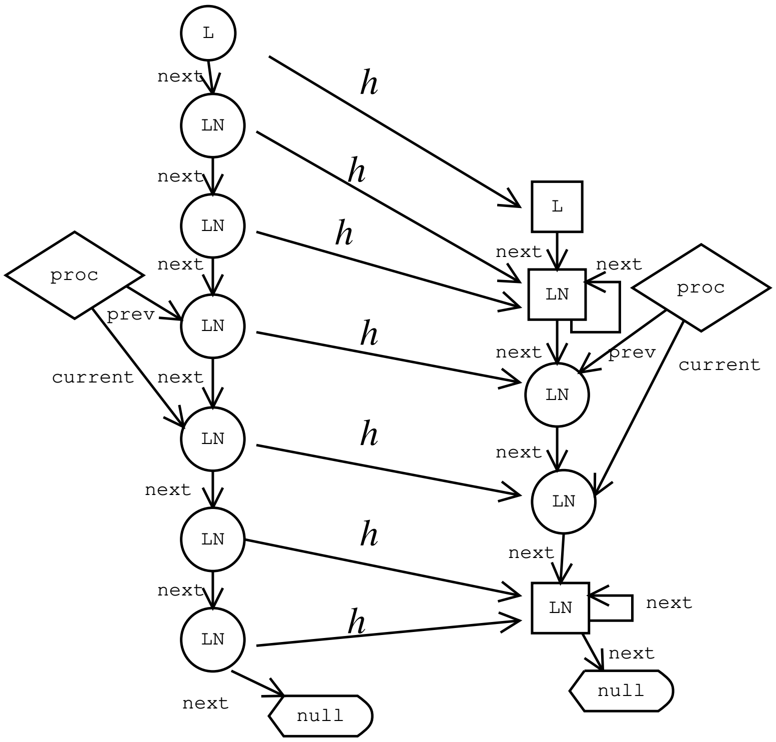


Figure 4-1: Abstraction Relation

```
role L { // List header
  fields first : LN | null;
}
role LN { // List node
  fields next : LN | null;
  slots LN.next | L.first;
  acyclic next;
}
```

Figure 4-1 shows a role graph and one of the concrete heaps represented by the role graph via homomorphism h. There are two local variables, `prev` and `current`, referencing distinct onstage objects. Onstage objects are isomorphic to onstage nodes in the role graph. In contrast, there are two objects mapped to each of the summary nodes with role `LN` (shown as `LN`-labelled rectangles in Figure 4-1). Note that the sets of objects mapped to these two summary nodes are disjoint. The first summary `LN`-node represents objects stored in the list before the object referenced by `prev`. The second summary `LN`-node represents objects stored in the list after the object referenced by `current`. $\triangle$

4.2 Transfer Functions

The key complication in developing the transfer functions for the role analysis is to accurately model the movement of objects onstage and offstage. For example, a load statement `x=y.f` may cause the object referred to by `y.f` to move onstage. In addition, if `x` was the only reference to an onstage object o before the statement executed, object o moves offstage after the execution of the load statement, and thus must satisfy the $\mathsf{locallyConsistent}$ predicate.

The analysis uses an expansion relation $\preceq$ to model the movement of objects onstage and a contraction relation $\succeq$ to model the movement of objects offstage. The expansion relation uses the invariant that offstage nodes have correct roles to generate possible aliasing relationships for the node being pulled onstage. The contraction relation establishes the role invariants for the node going offstage, allowing the node to be merged into the other offstage nodes and represented more compactly.

We present our role analysis as an abstract execution relation $\overset{\mathsf{st}}{\leadsto}$. The abstract execution ensures that the abstraction relation α is a forward simulation relation [63] from the space of concrete heaps with role assignments to the set $\mathsf{RoleGraphs}$. The simulation relation implies that the traces of $\leadsto$ include the traces of the instrumented semantics $\rightarrow$. To ensure that the program does not violate constraints associated with roles, it is thus sufficient to guarantee that $\bot_G$ is not reachable via $\leadsto$.

To prove that $\bot_G$ is not reachable in the abstract execution, the analysis computes for every program point p a set of role graphs $\mathcal{G}$ that conservatively approximates the possible program states at point p. The transfer function for a statement st is an image $[\![\mathsf{st}]\!](\mathcal{G}) = \{G' \mid G \in \mathcal{G}, G \overset{\mathsf{st}}{\leadsto} G'\}$. The analysis computes the relation $\overset{\mathsf{st}}{\leadsto}$ in three steps:

$$
\begin{tikzcd}
& \langle H_c, \rho_c \rangle \arrow[rr] \arrow[dl, "\alpha"'] \arrow[d, "\alpha"] & & \langle H'_c, \rho'_c \rangle \arrow[dr, "\alpha"] & \\
G_1 \arrow[r, phantom, "\preceq"] & G_2 \arrow[rr, Rightarrow, "\mathtt{st}"] & & G_3 \arrow[r, phantom, "\succeq"] & G_4
\end{tikzcd}
$$

Figure 4-2: Simulation Relation Between Abstract and Concrete Execution

Transition	Definition	Conditions
$\langle H, \rho, K \rangle \overset{\mathtt{x=y.f}}{\rightsquigarrow} G'$	$\langle H, \rho, K \rangle \overset{n_y, f}{\preceq} G_1 \overset{\mathtt{x=y.f}}{\Longrightarrow} G_2 \overset{n_x}{\succeq} G'$	$\langle \mathsf{proc}, \mathtt{x}, n_x \rangle, \langle \mathsf{proc}, \mathtt{y}, n_y \rangle \in H$
$\langle H, \rho, K \rangle \overset{\mathtt{x=y}}{\rightsquigarrow} G'$	$\langle H, \rho, K \rangle \overset{\mathtt{x=y}}{\Longrightarrow} G_1 \overset{n_1}{\succeq} G'$	$\langle \mathsf{proc}, \mathtt{x}, n_1 \rangle \in H$
$\langle H, \rho, K \rangle \overset{\mathtt{x=new}}{\rightsquigarrow} G'$	$\langle H, \rho, K \rangle \overset{\mathtt{x=new}}{\Longrightarrow} G_1 \overset{n_1}{\succeq} G'$	$\langle \mathsf{proc}, \mathtt{x}, n_1 \rangle \in H$
$\langle H, \rho, K \rangle \overset{\mathtt{st}}{\rightsquigarrow} G'$	$\langle H, \rho, K \rangle \overset{\mathtt{st}}{\Longrightarrow} G'$	$\mathsf{st} \in \{$`x.f=y`, `test(c)`, `setRole(x:r)`, `roleCheck(`$x_{1..p}$, ra`)`$\}$

Figure 4-3: Abstract Execution $\rightsquigarrow$

1. ensure that the relevant nodes are instantiated using expansion relation $\preceq$ (Section 4.2.1);

2. perform symbolic execution $\overset{\mathtt{st}}{\Longrightarrow}$ of the statement `st` (Section 4.2.3);

3. merge nodes if needed using contraction relation $\succeq$ to keep the role graph bounded (Section 4.2.2).

Figure 4-2 shows how the abstraction relation α relates $\preceq$, $\overset{\mathtt{st}}{\Longrightarrow}$, and $\succeq$ with the concrete execution $\rightarrow$ in instrumented semantics. Assume that a concrete heap $\langle H_c, \rho_c \rangle$ is represented by the role graph G_1. Then one of the role graphs G_2 obtained after expansion remains an abstraction of $\langle H_c, \rho_c \rangle$. The symbolic execution $\overset{\mathtt{st}}{\Longrightarrow}$ followed by the contraction relation $\succeq$ corresponds to the instrumented operational semantics $\rightarrow$.

Figure 4-3 shows rules for the abstract execution relation $\overset{\mathtt{st}}{\rightsquigarrow}$. Only Load statement uses the expansion relation, because the other statements operate on objects that are already onstage. Load, Copy, and New statements may remove a local variable reference from an object, so they use contraction relation to move the object offstage if needed. For the rest of the statements, the abstract execution reduces to symbolic execution $\Longrightarrow$ described in Section 4.2.3.

Transition	Definition	Condition
$\langle H,\rho,K\rangle \overset{n,f}{\preceq} \langle H,\rho,K\rangle$		$\langle n,f,n'\rangle \in H, n' \in \mathsf{onstage}(H)$
$\langle H,\rho,K\rangle \overset{n,f}{\preceq} G'$	$\langle H,\rho,K\rangle \underset{n'}{\overset{n_0}{\Uparrow}} \langle H_1,\rho_1,K_1\rangle \overset{n_0}{\parallel} G'$	$\langle n,f,n'\rangle \in H, n' \in \mathsf{offstage}(H)$ $\langle n,f,n_0\rangle \in H_1$

Figure 4-4: Expansion Relation

Nondeterminism and Failure

The $\overset{\mathtt{st}}{\leadsto}$ relation is not a function because the expansion relation $\preceq$ can generate a set of role graphs from a single role graph. Also, there might be no $\overset{\mathtt{st}}{\leadsto}$ transitions originating from a given state G if the symbolic execution $\Longrightarrow$ produces no results. This corresponds to a trace which cannot be extended further due to a `test` statement which fails in state G. This is in contrast to a transition from G to $\bot_G$ which indicates a potential role consistency violation or a null pointer dereference. We assume that $\Longrightarrow$ and $\succeq$ relations contain the transition $\langle \bot_G, \bot_G\rangle$ to propagate the error role graph. In most cases we do not show the explicit transitions to error states.

4.2.1 Expansion

Figure 4-4 shows the expansion relation $\overset{n,f}{\preceq}$. Given a role graph $\langle H,\rho,K\rangle$, expansion attempts to produce a set of role graphs $\langle H',\rho',K'\rangle$ in each of which $\langle n,f,n_0\rangle \in H'$ and $K(n_0) = i$. Expansion is used in abstract execution of the Load statement. It first checks for null pointer dereference and reports an error if the check fails. If $\langle n,f,n'\rangle \in H$ and $K(n') = i$ already hold, the expansion returns the original state. Otherwise, $\langle n,f,n'\rangle \in H$ with $K(n') = s$. In that case, the summary node n' is first instantiated using instantiation relation $\underset{n'}{\overset{n_0}{\Uparrow}}$. Next, the split relation $\overset{n_0}{\parallel}$ is applied. Let $\rho(n_0) = r$. The split relation ensures that n_0 is not a member of any cycle of offstage nodes which contains only edges in $\mathsf{acyclic}(r)$. We explain instantiation and split in more detail below.

Instantiation

Figure 4-5 presents the instantiation relation. Given a role graph $G = \langle H,\rho,K\rangle$, instantiation $\underset{n'}{\overset{n_0}{\Uparrow}}$ generates the set of role graphs $\langle H',\rho',K'\rangle$ such that each concrete heap represented by $\langle H,\rho,K\rangle$ is represented by one of the graphs $\langle H',\rho',K'\rangle$. Each of the new role graphs contains a fresh individual node n_0 that satisfies $\mathsf{localCheck}$. The edges of n_0 are a subset of edges from and to n'.

Let H_0 be a subset of the references between n' and onstage nodes, and let H_1 be a subset of the references between n' and offstage nodes. References in H_0 are moved from n' to the new node n_0, because they represent at most one reference, while

$\langle H, \rho, K\rangle \overset{n_0}{\underset{n'}{\Uparrow}} \langle H', \rho', K'\rangle$	$H' = H \setminus H_0 \cup H_0' \cup H_1'$ $\rho' = \rho[n_0 \mapsto \rho(n')]$ $K' = K[n_0 \mapsto i]$ $\mathsf{localCheck}(n_0, \langle H', \rho', K'\rangle)$ $H_0 \subseteq H \cap (\mathsf{onstage}(H) \times F \times \{n'\} \cup \{n'\} \times F \times \mathsf{onstage}(H))$ $H_1 \subseteq H \cap (\mathsf{offstage}(H) \times F \times \{n'\} \cup \{n'\} \times F \times \mathsf{offstage}(H))$ $H_0' = \mathsf{swing}(n', n_0, H_0)$ $H_1' \subseteq \mathsf{swing}(n', n_0, H_1)$

$$\begin{array}{rcl} \mathsf{swing}(n_{\mathsf{old}}, n_{\mathsf{new}}, H) & = & \{\langle n_{\mathsf{new}}, f, n\rangle \mid \langle n_{\mathsf{old}}, f, n\rangle \in H\} \cup \\ & & \{\langle n, f, n_{\mathsf{new}}\rangle \mid \langle n, f, n_{\mathsf{old}}\rangle \in H\} \cup \\ & & \{\langle n_{\mathsf{new}}, f, n_{\mathsf{new}}\rangle \mid \langle n_{\mathsf{old}}, f, n_{\mathsf{old}}\rangle \in H\} \end{array}$$

Figure 4-5: Instantiation Relation

references in H_1 are copied to n_0 because they may represent multiple concrete heap references. Moving a reference is formalized via the swing operation in Figure 4-5.

The instantiation of a single graph can generate multiple role graphs depending on the choice of H_0' and H_1'. The number of graphs generated is limited by the existing references of node n' and by the localCheck requirement for n_0. This is where our role analysis takes advantage of the constraints associated with role definitions to reduce the number of aliasing possibilities that need to be considered.

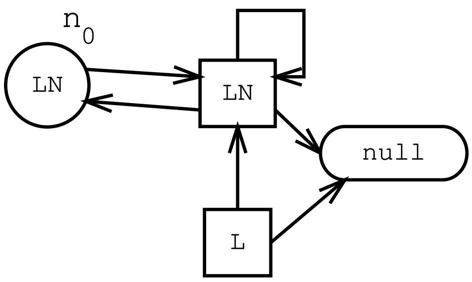
Split

The split relation is important for verifying operations on data structures such as skip lists and sparse matrices. It is also useful for improving the precision of the initial set of role graphs on procedure entry (Section 5.2.1).

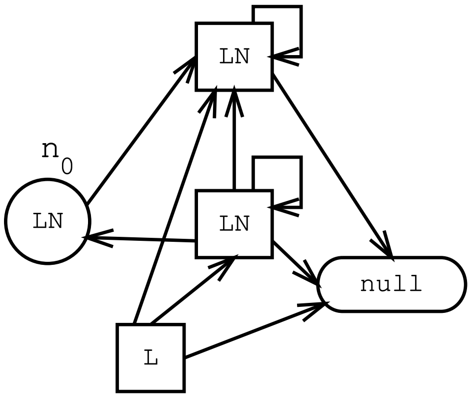
The goal of the split relation is to exploit the acyclicity constraints associated with role definitions. After a node n_0 is brought onstage, split represents the acyclicity condition of $\rho(n_0)$ explicitly by eliminating impossible paths in the role graph. It uses additional offstage nodes to encode the reachability information implied by the acyclicity conditions. This information can then be used even after the role of node n_0 changes. In particular, it allows the acyclicity condition of n_0 to be verified when n_0 moves offstage.

Example 24 Consider a role graph for an acyclic list with nodes `LN` and a header node `L`. The instantiated node n_0 is in the middle of the list. Figure 4-6 a) shows a role graph with a single summary node representing all offstage `LN`-nodes. Figure 4-6 b) shows the role graph after applying the split relation. The resulting role graph contains two `LN` summary nodes. The first `LN` summary node represents objects definitely reachable from n_0 along `next` edges; the second summary `NL` node represents objects definitely not reachable from n_0. $\triangle$

Figure 4-7 shows the definition of the split operation on node n_0, denoted by $\overset{n_0}{\|}$. Let $G = \langle H, \rho, K\rangle$ be the initial role graph and $\rho(n_0) = r$. If $\mathsf{acyclic}(r) = \emptyset$, then the



a) Before Split



b) After Split

Figure 4-6: A Role Graph for an Acyclic List

$$\langle H, \rho, K\rangle \overset{n_0}{\parallel} \langle H, \rho, K\rangle, \quad \mathsf{acycCheck}(n_0, \langle H, \rho, K\rangle, \mathsf{offstage}(H))$$
$$\langle H, \rho, K\rangle \overset{n_0}{\parallel} \langle H', \rho', K'\rangle, \quad \neg\mathsf{acycCheck}(n_0, \langle H, \rho, K\rangle, \mathsf{offstage}(H))$$

where

$$\begin{array}{l}
H' = (H \setminus H_{\mathrm{cyc}}) \cup H_{\mathrm{off}} \cup B_{\mathrm{fNR}} \cup B_{\mathrm{fR}} \cup B_{\mathrm{tNR}} \cup B_{\mathrm{tR}} \cup N_{\mathrm{f}} \cup N_{\mathrm{t}} \\
H_{\mathrm{cyc}} = \{\langle n_1, f, n_2\rangle \mid n_1 \text{ or } n_2 \in S_{\mathrm{cyc}}\} \\
H_{\mathrm{off}} \;=\; \{\, \langle n_1', f, n_2'\rangle \;\mid\; n_1 = c(n_1'), n_2 = c(n_2'), \\
\qquad\qquad\qquad\qquad\quad n_1, n_2 \in \mathsf{offstage}_1(H), n_1 \text{ or } n_2 \in S_{\mathrm{cyc}}, \\
\qquad\qquad\qquad\qquad\quad \langle n_1, f, n_2\rangle \in H \,\} \\
\qquad\quad \setminus (S_{\mathrm{R}} \times \mathsf{acyclic}(r) \times S_{\mathrm{NR}}) \\
H \cap (\mathsf{onstage}(H) \times F \cup \{n_0\} \times \mathsf{acyclic}(r)) \times S_{\mathrm{cyc}} = A_{\mathrm{fNR}} \uplus A_{\mathrm{fR}} \\
H \cap S_{\mathrm{cyc}} \times (\mathsf{acyclic}(r) \times \{n_0\} \cup F \times \mathsf{onstage}(H)) = A_{\mathrm{tNR}} \uplus A_{\mathrm{tR}} \\
B_{\mathrm{fNR}} = \{\langle n_1, f, h_{\mathrm{NR}}(n_2)\rangle \mid \langle n_1, f, n_2\rangle \in A_{\mathrm{fNR}}\} \\
B_{\mathrm{fR}} = \{\langle n_1, f, h_{\mathrm{R}}(n_2)\rangle \mid \langle n_1, f, n_2\rangle \in A_{\mathrm{fR}}\} \\
B_{\mathrm{tNR}} = \{\langle h_{\mathrm{NR}}(n_1), f, n_2\rangle \mid \langle n_1, f, n_2\rangle \in A_{\mathrm{tNR}}\} \\
B_{\mathrm{tR}} = \{\langle h_{\mathrm{R}}(n_1), f, n_2\rangle \mid \langle n_1, f, n_2\rangle \in A_{\mathrm{tR}}\} \\
N_{\mathrm{f}} = \{\langle n_0, f, n'\rangle \mid n' \in S_{\mathrm{R}}, \langle n_0, f, c(n')\rangle \in H, f \in \mathsf{acyclic}(r)\} \\
N_{\mathrm{t}} = \{\langle n', f, n_0\rangle \mid n' \in S_{\mathrm{NR}}, \langle c(n'), f, n_0\rangle \in H, f \in \mathsf{acyclic}(r)\} \\
S_{\mathrm{cyc}} = \{n \mid \exists n_1, \ldots, n_{p-1} \in \mathsf{offstage}(H) : \\
\qquad \langle n_0, f_0, n_1\rangle, \ldots, \langle n_k, f_k, n\rangle, \langle n, f_{k+1}, n_{k+2}\rangle, \langle n_{p-1}, f_{p-1}, n_0\rangle \in H, \\
\qquad f_0, \ldots, f_{p-1} \in \mathsf{acyclic}(r)\} \\
\mathsf{offstage}_1(H) = \mathsf{offstage}(H) \setminus \{n_0\} \\
r = \rho(n_0) \\
\\
\rho'(c(n)) = \rho(n) \\
K'(c(n)) = K(n)
\end{array}$$

Figure 4-7: Split Relation

split operation returns the original graph G; otherwise it proceeds as follows. Call a path in graph H *cycle-inducing* if all of its nodes are offstage and all of its edges are in $\mathsf{acyclic}(r)$. Let S_{cyc} be the set of nodes n such that there is a cycle-inducing path from n_0 to n and a cycle-inducing path from n to n_0.

The goal of the split operation is to split the set S_{cyc} into a fresh set of nodes S_{NR} representing objects definitely not reachable from n_0 along edges in $\mathsf{acyclic}(r)$ and a fresh set of nodes S_{R} representing objects definitely reachable from n_0. Each of the newly generated graphs H' has the following properties:

1) merging the corresponding nodes from S_{NR} and S_{R} in H' yields the original graph H;
2) n_0 is not a member of any cycle in H' consisting of offstage nodes and edges in $\mathsf{acyclic}(r)$;
3) onstage nodes in H' have the same number of fields and aliases as in H.

Let $S_0 = \mathsf{nodes}(H) \setminus S_{\mathrm{cyc}}$ and let $h_{\mathrm{NR}} : S_{\mathrm{cyc}} \to S_{\mathrm{NR}}$ and $h_{\mathrm{R}} : S_{\mathrm{cyc}} \to S_{\mathrm{R}}$ be bijections. Define a function $c : \mathsf{nodes}(H') \to \mathsf{nodes}(H)$ as follows:

$$c(n) = \begin{cases} n, & n \in S_0 \\ h_{\mathrm{R}}^{-1}(n), & n \in S_{\mathrm{R}} \\ h_{\mathrm{NR}}^{-1}(n), & n \in S_{\mathrm{NR}} \end{cases}$$

Then $H' \subseteq \{\langle n_1', f, n_2' \rangle \mid \langle c(n_1'), f, c(n_2') \rangle \in H\}$.

Because there are two copies of S_0 in H', there might be multiple edges $\langle n_1', f, n_2' \rangle$ in H' corresponding to an edge $\langle c(n_1), f, c(n_2) \rangle \in H$.

If both n_1' and n_2' are offstage nodes other than n_0, we always include $\langle n_1', f, n_2' \rangle$ in H' unless $\langle n_1', f, n_2' \rangle \in S_{\mathrm{R}} \times \mathsf{acyclic}(r) \times S_{\mathrm{NR}}$. The last restriction prevents cycles in H'.

For an edge $\langle n_1, f, n_2 \rangle \in H$ where $n_1 \in \mathsf{onstage}(H)$ and $n_2 \in S_{\mathrm{cyc}}$ we include in H' either the edge $\langle n_1, f, h_{\mathrm{NR}}(n_2) \rangle$ or $\langle n_1, f, h_{\mathrm{R}}(n_2) \rangle$ but not both. Split generates multiple graphs H' to cover both cases. We proceed analogously if $n_2 \in \mathsf{onstage}(H)$ and $n_1 \in S_{\mathrm{cyc}}$. The node n_0 itself is treated in the same way as onstage nodes for $f \notin \mathsf{acyclic}(r)$. If $f \in \mathsf{acyclic}(r)$ then we choose references *to* n_0 to have a source in S_{NR}, whereas the reference *from* n_0 have the target in S_{R}.

Details of the split construction are given in Figure 4-7. The intuitive meaning of the sets of edges is the following:

H_{off} : edges between offstage nodes
B_{fNR} : edges from onstage nodes to S_{NR}
B_{fR} : edges from onstage nodes to S_{R}
B_{tNR} : edges from S_{NR} to onstage nodes
B_{tR} : edges from S_{R} to onstage nodes
N_{f} : $\mathsf{acyclic}(r)$-edges from n_0 to S_{R}
N_{t} : $\mathsf{acyclic}(r)$-edges from S_{NR} to n_0

The sets B_{fNR} and B_{fR} are created as images of the sets A_{fNR} and A_{fR} which partition edges from onstage nodes to nodes in S_{cyc}. Similarly, the sets B_{tNR} and B_{tR} are

$\langle H, \rho, K \rangle \overset{n}{\succeq} \langle H, \rho, K \rangle$	$\exists \mathsf{x} \in \mathsf{var}(\mathsf{proc}) :$ $\langle \mathsf{proc}, \mathsf{x}, n \rangle \in H$
$\langle H, \rho, K \rangle \overset{n}{\succeq} \mathsf{normalize}(\langle H, \rho, K \rangle)$	$\mathsf{nodeCheck}(n, \langle H, \rho, K \rangle, \mathsf{offstage}(H))$

Figure 4-8: Contraction Relation

$$\mathsf{normalize}(\langle H, \rho, K \rangle) = \langle H', \rho', K' \rangle$$

$$
\begin{aligned}
\text{where}\quad & H' = \{\langle n_{1/\sim}, f, n_{2/\sim} \rangle \mid \langle n_1, f, n_2 \rangle \in H\} \\
& \rho'(n_{/\sim}) = \rho(n) \\
& K'(n_{/\sim}) = \begin{cases} i, & n_{/\sim} = \{n\}, K(n) = i \\ s, & \text{otherwise} \end{cases} \\
& n_1 \sim n_2 \text{ iff } n_1 = n_2 \text{ or} \\
& \qquad (n_1, n_2 \in \mathsf{offstage}(H), \rho(n_1) = \rho(n_2), \\
& \qquad \forall n_0 \in \mathsf{onstage}(H) : (\mathsf{reach}(n_0, n_1) \text{ iff } \mathsf{reach}(n_0, n_2)) \\
& \mathsf{reach}(n_0, n) \text{ iff } \exists n_1, \ldots, n_{p-1} \in \mathsf{offstage}(n), \exists f_1, \ldots, f_p \in \mathsf{acyclic}(\rho(n_0)) : \\
& \qquad \langle n_0, f_1, n_1 \rangle, \ldots, \langle n_{p-1}, f_p, n \rangle \in H
\end{aligned}
$$

Figure 4-9: Normalization

created as images of the sets A_{tNR} and A_{tR} which partition edges from nodes in S_{cyc} to onstage nodes.

We note that if in the split operation $S_{\text{cyc}} = \emptyset$ then split has no effect and need not be performed. In Figure 4-6, after performing a single split, there is no need to split for subsequent elements of the list. Examples like this indicate that split will not be invoked frequently during the analysis.

4.2.2 Contraction

Figure 4-8 shows the non-error transitions of the contraction relation $\overset{n}{\succeq}$. The analysis uses contraction when a reference to node n is removed. If there are other references to n, the result is the original graph. Otherwise n has just gone offstage, so the analysis invokes nodeCheck. If the check fails, the result is $\bot_G$. If the role check succeeds, the contraction invokes normalization operation to ensure that the role graph remains bounded. For simplicity, we use normalization whenever nodeCheck succeeds, although it is sufficient to perform normalization only at program points adjacent to back edges of the control-flow graph.

Normalization

Figure 4-9 shows the normalization relation. Normalization accepts a role graph $\langle H, \rho, K \rangle$ and produces a normalized role graph $\langle H', \rho', K' \rangle$ which is a factor graph

Statement s	Transition	Conditions
`x = y.f`	$\langle H \uplus \{\mathsf{proc}, \mathtt{x}, n_x\}, \rho, K\rangle \stackrel{\mathtt{st}}{\Longrightarrow} \langle H \uplus \{\mathsf{proc}, \mathtt{x}, n_f\}, \rho, K\rangle$	$\langle \mathsf{proc}, \mathtt{y}, n_y\rangle, \langle n_y, f, n_f\rangle \in H$
`x.f = y`	$\langle H \uplus \{n_x, f, n_f\}, \rho, K\rangle \stackrel{\mathtt{st}}{\Longrightarrow} \langle H \uplus \{n_x, f, n_y\}, \rho, K\rangle$	$\langle \mathsf{proc}, \mathtt{x}, n_x\rangle, \langle \mathsf{proc}, \mathtt{y}, n_y\rangle \in H$ $n_f \in \mathsf{onstage}(H)$
`x = y`	$\langle H \uplus \{\mathsf{proc}, \mathtt{x}, n_x\}, \rho, K\rangle \stackrel{\mathtt{st}}{\Longrightarrow} \langle H \uplus \{\mathsf{proc}, \mathtt{x}, n_y\}, \rho, K\rangle$	$\langle \mathsf{proc}, \mathtt{y}, n_y\rangle \in H$
`x = new`	$\langle H \uplus \{\mathsf{proc}, \mathtt{x}, n_x\}, \rho, K\rangle \stackrel{\mathtt{st}}{\Longrightarrow} \langle H \uplus \{\mathsf{proc}, \mathtt{x}, n_n\}, \rho', K\rangle$	n_n fresh $\rho' = \rho[n_n \mapsto \mathsf{unknown}]$
`test(c)`	$\langle H, \rho, K\rangle \stackrel{\mathtt{st}}{\Longrightarrow} \langle H, \rho, K\rangle$	$\mathsf{satisfied}(\mathsf{c}, H)$
`setRole(x:r)`	$\langle H, \rho, K\rangle \stackrel{\mathtt{st}}{\Longrightarrow} \langle H, \rho[n_x \mapsto \mathtt{r}], K\rangle$	$\langle \mathsf{proc}, \mathtt{x}, n_x\rangle \in H$ $\mathsf{roleChOk}(n_x, \mathtt{r}, \langle H, \rho, K\rangle)$
`roleCheck`$(x_{1..p}, \mathsf{ra})$	$\langle H, \rho, K\rangle \stackrel{\mathtt{st}}{\Longrightarrow} \langle H, \rho, K\rangle$	$\forall i\ \langle \mathsf{proc}, \mathtt{x}_i, n_i\rangle \in H$ $\mathsf{nodeCheck}(n_i, \langle H, \rho, K\rangle, S)$ $S = \mathsf{offstage}(H) \cup \{n_i\}_i$ $\rho(n_i) = \mathsf{ra}(n_i)$

$$\mathsf{satisfied}(\mathtt{x{==}y}, H_c) \text{ iff } \{o \mid \langle \mathsf{proc}, \mathtt{x}, o\rangle \in H_c\} = \{o \mid \langle \mathsf{proc}, \mathtt{y}, o\rangle \in H_c\}$$

$$\mathsf{satisfied}(\mathtt{!(x{==}y)}, H_c) \text{ iff not } \mathsf{satisfied}(\mathtt{x{==}y}, H_c)$$

Figure 4-10: Symbolic Execution of Basic Statements

of $\langle H, \rho, K\rangle$ under the equivalence relation $\sim$. Two offstage nodes are equivalent under $\sim$ if they have the same role and the same reachability from onstage nodes. Here we consider node n to be reachable from an onstage node n_0 iff there is some path from n_0 to n whose edges belong to $\mathsf{acyclic}(\rho(n_0))$ and whose nodes are all in $\mathsf{offstage}(H)$. Note that, by construction, normalization avoids merging nodes which were previously generated in the split operation $\|$, while still ensuring a bound on the size of the role graph. For a procedure with l local variables, f fields and r roles the number of nodes in a role graph is on the order of $r2^l$ so the maximum size of a chain in the lattice is of the order of 2^{r2^l}. To ensure termination we consider role graphs equal up to isomorphism. Isomorphism checking can be done efficiently if normalization assigns canonical names to the equivalence classes it creates.

4.2.3 Symbolic Execution

Figure 4-10 shows the symbolic execution relation $\stackrel{\mathtt{st}}{\Longrightarrow}$. In most cases, the symbolic execution of a statement acts on the abstract heap in the same way that the statement would act on the concrete heap. In particular, the Store statement always performs strong updates. The simplicity of symbolic execution is due to conditions 3) and 5) in the abstraction relation α. These conditions are ensured by the $\preceq$ relation which instantiates nodes, allowing strong updates. The symbolic execution also verifies the consistency conditions that are not verified by $\preceq$ or $\succeq$.

Verifying Reference Removal Consistency

The abstract execution $\stackrel{\mathtt{st}}{\rightsquigarrow}$ for the Store statement can easily verify the Store safety condition from section 3.4.2, because the set of onstage and offstage nodes is known precisely for every role graph. It returns $\perp_G$ if the safety condition fails.

Symbolic Execution of setRole

The `setRole(x:r)` statement sets the role of node n_x referenced by variable `x` to `r`. Let $G = \langle H, \rho, K\rangle$ be the current role graph and let $\langle \mathsf{proc}, \mathtt{x}, n_x\rangle \in H$. If n_x has no adjacent offstage nodes, the role change always succeeds. In general, there are restrictions on when the change can be done. Let $\langle H_c, \rho_c\rangle$ be a concrete heap with role assignment represented by G and h be a homomorphism from H_c to H. Let $h(o_x) = n_x$. Let $r_0 = \rho_c(o_x)$. The symbolic execution must make sure that the condition $\mathsf{conW}(\rho_c, H_c, \mathsf{offstage}(H_c))$ continues to hold after the role change. Because the set of onstage nodes does not change, it suffices to ensure that the original roles for offstage nodes are consistent with the new role r. The acyclicity constraint involves only offstage nodes, so it remains satisfied. The other role constraints are local, so they can only be violated for offstage neighbors of n_x. To make sure that no violations occur, we require:

1. $\mathtt{r} \in \mathsf{field}_f(\rho(n))$ for all $\langle n, f, n_x\rangle \in H$, and
2. $\langle \mathtt{r}, f\rangle \in \mathsf{slot}_i(\rho(n))$ for all $\langle n_x, f, n\rangle \in H$ and every slot i such that $\langle r_0, f\rangle \in \mathsf{slot}_i(\rho(n))$

This is sufficient to guarantee $\mathsf{conW}(\rho_c, H_c, \mathsf{offstage}(H_c))$. To ensure condition 2) in Definition 22 of the abstraction relation, we require that for every $\langle f, g\rangle \in \mathsf{identities}(\mathtt{r})$,

1. $\langle f, g\rangle \in \mathsf{identities}(r_0)$ or
2. for all $\langle n_x, f, n\rangle \in H$: $K(n) = i$ and ($\langle n, g, n'\rangle \in H$ implies $n' = n_x$).

Symbolic Execution of roleCheck

The symbolic execution of the statement $\mathtt{roleCheck}(x_1, \ldots, x_p, \mathsf{ra})$ ensures that the conW predicate of the concrete semantics is satisfied for the concrete heaps which correspond to the current abstract role graph. The symbolic execution returns the error graph $\perp_G$ if ρ is inconsistent with ra or if any of the nodes n_i referenced by x_i fail to satisfy $\mathsf{nodeCheck}$.

Accessibility Condition

The analysis ensures that the accessibility condition for the Load statement will be satisfied in procedure proc before procedure proc is called. This technique makes use of procedure effects and is described in Chapter 5.

4.2.4 Node Check

The analysis uses the nodeCheck predicate to incrementally maintain the abstraction relation. We first define the predicate localCheck, which roughly corresponds to the predicate locallyConsistent (Definition 2), but ignores the nonlocal acyclicity condition and additionally ensures condition 2) from Definition 22.

Definition 25 *For a role graph $G = \langle H, \rho, K \rangle$, an individual node n and a set S, the predicate* $\mathsf{localCheck}(n, G)$ *holds iff the following conditions are met. Let $r = \rho(n)$.*

1A. *(Outgoing fields check) For fields $f \in F$, if $\langle n, f, n' \rangle \in H$ then $\rho(n') \in \mathsf{field}_f(r)$.*

2A. *(Incoming slots check) Let $\{\langle n_1, f_1 \rangle, \ldots, \langle n_k, f_k \rangle\} = \{\langle n', f \rangle \mid \langle n', f, n \rangle \in H\}$ be the set of all aliases of node n in abstract heap H. Then $k = \mathsf{slotno}(r)$ and there exists a permutation p of the set $\{1, \ldots, k\}$ such that $\langle \rho(n_i), f_i \rangle \in \mathsf{slot}_{p_i}(r)$ for all i.*

3A. *(Identity Check) If $\langle n, f, n' \rangle \in H$, $\langle n', g, n'' \rangle \in H$, $\langle f, g \rangle \in \mathsf{identities}(r)$, and $K(n') = i$, then $n = n''$.*

4A. *(Neighbor Identity Check) For every edge $\langle n', f, n \rangle \in H$, if $K(n') = i$, $\rho(n') = r'$ and $\langle f, g \rangle \in \mathsf{identities}(r')$ then $\langle n, g, n' \rangle \in H$.*

5A. *(Field Sanity Check) For every $f \in F$ there is exactly one edge $\langle n, f, n' \rangle \in H$.*

Conditions 1A and 2A correspond to conditions 1) and 2) in Definition 2. Condition 3) in Definition 19 is not necessarily implied by condition 3A) if some of the neighbors of n are summary nodes. Condition 3) cannot be established based only on summary nodes, because verifying an identity constraint for field f of node n where $\langle n, f, n' \rangle \in H$ requires knowing the identity of n', not only its existence and role. We therefore rely on Condition 2) of the Definition 22 to ensure that identity relations of neighbors of node n are satisfied before n moves offstage.

The predicate $\mathsf{acycCheck}(n, G, S)$ verifies the acyclicity condition from Definition 19.

Definition 26 *We say that node $n \in \mathsf{nodes}(H)$ satisfies an acyclicity check in graph $G = \langle H, \rho, K \rangle$ with respect to set S, and we write $\mathsf{acycCheck}(n, G, S)$, iff it is not the case that H contains a cycle $n_1, f_1, \ldots, n_s, f_s, n_1$ where $n_1 = n$, $f_1, \ldots, f_s \in \mathsf{acyclic}(\rho(n))$ and $n_1, \ldots, n_s \in S$.*

This enables us to define the $\mathsf{nodeCheck}$ predicate.

Definition 27 $\mathsf{nodeCheck}(n, G, S)$ *holds iff both the predicate* $\mathsf{localCheck}(n, G)$ *and the predicate* $\mathsf{acycCheck}(n, G, S)$ *hold.*

Chapter 5

Interprocedural Role Analysis

This chapter describes the interprocedural aspects of our role analysis. Interprocedural role analysis can be viewed as an instance of the functional approach to interprocedural data-flow analysis [80]. For each program point p, the role analysis approximates program traces from procedure entry to point p. The solution in [80] proposes tagging the entire data-flow fact G at point p with the data flow fact G_0 at procedure entry. In contrast, our analysis computes the correspondence between the heaps at procedure entry and the heaps at point p at the granularity of sets of objects that constitute the role graphs. This allows our analysis to detect which regions of the heap have been modified. We approximate the concrete executions of a procedure with *procedure transfer relations* consisting of 1) an initial context and 2) a set of *effects*. Effects are fine-grained transfer relations which summarize load and store statements and can naturally describe local heap modifications. In this work we assume that procedure transfer relations are supplied and we are concerned with a) verifying that transfer relations are a conservative approximation of procedure implementation b) instantiating transfer relations at call sites.

5.1 Procedure Transfer Relations

A transfer relation for a procedure proc extends the procedure signature with an initial context denoted $\mathsf{context}(\mathsf{proc})$, and procedure effects denoted $\mathsf{effect}(\mathsf{proc})$.

5.1.1 Initial Context

Figures 5-1 and 5-2 contain examples of initial context specification. An initial context is a description of the initial role graph $\langle H_{\mathsf{IC}}, \rho_{\mathsf{IC}}, K_{\mathsf{IC}} \rangle$ where ρ_{IC} and K_{IC} are determined by a `nodes` declaration and H_{IC} is determined by a `edges` declaration. The initial role graph specifies a set of concrete heaps at procedure entry and assigns names for sets of nodes in these heaps. The next definition is similar to Definition 22.

Definition 28 *We say that a concrete heap $\langle H_c, \rho_c \rangle$ is represented by the initial role graph $\langle H_{\mathsf{IC}}, \rho_{\mathsf{IC}}, K_{\mathsf{IC}} \rangle$ and write $\langle H_c, \rho_c \rangle \, \alpha_0 \langle H_{\mathsf{IC}}, \rho_{\mathsf{IC}}, K_{\mathsf{IC}} \rangle$, iff there exists a function h_0 : $\mathsf{nodes}(H_c) \to \mathsf{nodes}(H_{\mathsf{IC}})$ such that*

1. $\mathsf{conW}(\rho_c, H_c, h_0^{-1}(\mathsf{read}(\mathsf{proc}))$*;*

2. h_0 *is a graph homomorphism;*

3. $K_{\mathsf{IC}}(n) = i$ *implies* $|h_0^{-1}(n)| \leq 1$*;*

4. $h_0(\mathsf{null}_c) = \mathsf{null}$ *and* $h_0(\mathsf{proc}_c) = \mathsf{proc}$*;*

5. $\rho_c(o) = \rho_{\mathsf{IC}}(h_0(o))$ *for every object* $o \in \mathsf{nodes}(H_c)$*.*

Here $\mathsf{read}(\mathsf{proc})$ is the set of initial-context nodes read by the procedure (see below). For simplicity, we assume one context per procedure; it is straightforward to generalize the treatment to multiple contexts.

A context is specified by declaring a list of nodes and a list of edges.

A list of nodes is given with `nodes` declaration. It specifies a role for every node at procedure entry. Individual nodes are denoted with lowercase identifiers, summary nodes with uppercase identifiers. By using summary nodes it is possible to indicate disjointness of entire heap regions and reachability between nodes in the heap.

There are two kinds of edges in the initial role graph: parameter edges and heap edges. A parameter edge `p->pn` is interpreted as $\langle \mathsf{proc}, \mathtt{p}, \mathtt{pn} \rangle \in H_{\mathsf{IC}}$. We require every parameter edge to have an individual node as a target, we call such node a *parameter node*. The role of a parameter node referenced by $\mathsf{param}_i(\mathsf{proc})$ is always $\mathsf{preR}_i(\mathsf{proc})$. Since different nodes in the initial role graph denote disjoint sets of concrete objects, parameter edges

```
p1 -> n1
p2 -> n1
```

imply that parameters `p1` and `p2` must be aliased,

```
p1 -> n1
p2 -> n2
```

force `p1` and `p2` to be unaliased, whereas

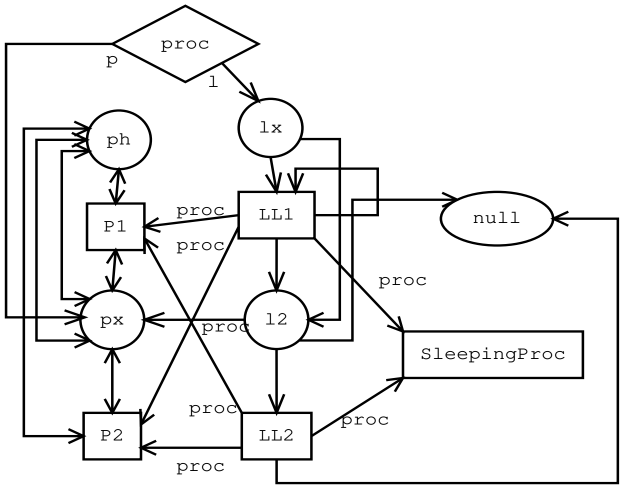
```
p1 -> n1|n2
p2 -> n1|n2
```

allow for both possibilities. A heap edge `n -f-> m` denotes $\langle \mathtt{n}, \mathtt{f}, \mathtt{m} \rangle \in H_{\mathsf{IC}}$. The shorthand notation

```
n1 -f-> n2
   -g-> n3
```

denotes two heap edges $\langle \mathtt{n1}, \mathtt{f}, \mathtt{n2} \rangle, \langle \mathtt{n1}, \mathtt{g}, \mathtt{n3} \rangle \in H_{\mathsf{IC}}$. An expression `n1 -f-> n2|n3` denotes two edges `n1 -f-> n2` and `n1 -f-> n3`. We use similar shorthands for parameter edges.

Example 29 Figure 5-1 shows an initial context graph for the `kill` procedure from Example 17. It is a refinement of the role reference diagram of Figure 1-1 as it gives description of the heap specific to the entry of `kill` procedure. The initial context




```
nodes ph : RunningHeader,
      P1, px, P2 : RunningProc,
      lx : LiveHeader,
      LL1, l2, LL2 : LiveList;
edges p-> px, l-> px,
      ph -next-> P1|px
         -prev-> px|P2,
      P1 -next-> P1|px
         -prev-> ph|P1,
      px -next-> P2|ph
         -prev-> P1|ph,
      P2 -next-> P2|ph
         -prev-> P2|px,
      lx -next-> LL1|l2,
      LL1 -next-> LL1|l2
          -proc-> P1|P2|SleepingProc
      l2 -next-> LL2|null
         -proc-> px,
      LL2 -next-> LL2|null
          -proc-> P1|P2|SleepingProc
```

Figure 5-1: Initial Context for `kill` Procedure makes explicit the fact that there is only one header node for the list of running processes (ph) and one header node for the list of all active processes (lx). More importantly, it shows that traversing the list of active processes reaches a node l2 whose proc field references the parameter node px. This is sufficient for the analysis to conclude that there will be no null pointer dereferences in the while loop of kill procedure since l2 is reached before null. △

We assume that the initial context always contains the role reference diagram RRD (Definition 8). Nodes from RRD are called *anonymous nodes* and are referred to via role name. This further reduces the size of initial context specifications by leveraging global role definitions. In Figure 5-1 there is no need to specify edges originating from SleepingProc or even mention the node SleepingTree, since role definitions alone contain enough information on this part of the heap to enable the analysis of the procedure.

5.1.2 Procedure Effects

Procedure effects conservatively approximate the region of the heap that the procedure accesses and indicate changes to the referencing relationships in that region. There are two kinds of effects: read effects and write effects.

A *read effect* specifies a set read(proc) of initial graph nodes accessed by the procedure. It is used to ensure that the accessibility condition in Section 3.4.3 is satisfied. If the set of nodes denoted by read(proc) is mapped to a node n which is onstage in the caller but is not an argument of the procedure call, a role check error is reported at the call site.

Write effects are used to modify caller's role graph to conservatively model the procedure call. A write effect $e_1.f = e_2$ approximates Store operations within a procedure. The expression e_1 denotes objects being written to, f denotes the field written, and e_2 denotes the set of objects which could be assigned to the field. Write effects are *may* effects by default, which means that the procedure is free not to perform them. It is possible to specify that a write effect *must* be performed by prefixing it with a "!" sign.

Example 30 In Figure 5-2, the insert procedure inserts an isolated cell into the end of an acyclic singly linked list. As a result, the role of the cell changes to LN. The initial context declares parameter nodes ln and xn (whose initial roles are deduced from roles of parameters), and mentions anonymous LN node from a default copy of the role reference diagram RRD. The code of the procedure is summarized with two write effects. The first write effect indicates that the procedure may perform zero or more Store operations to field next of nodes mapped to ln or LN in context(proc). The second write effect indicates that the execution of the procedure must perform a Store to the field next of xn node where the reference stored is either a node mapped onto anonymous LN node or null. △

Effects also describe assignments that procedures perform on the newly created nodes. Here we adopt a simple solution of using a single summary node denoted NEW

```
procedure insert(l : L,
                 x : IsolatedN ->> LN)
nodes ln, xn;
edges l-> ln, x-> xn,
      ln -next-> LN|null;
effects ln|LN . next = xn,
        ! xn.next = LN|null;
local c, p;
{
  p = l;
  c = l.next;
  while (c!=null) {
    p = c;
    c = p.next;
  }
  p.next = x;
  x.next = c;
  setRole(x:LN);
}
```

Figure 5-2: Insert Procedure for Acyclic List

to represent all nodes created inside the procedure. We write $\mathsf{nodes}_0(H_{\mathsf{IC}})$ for the set $\mathsf{nodes}(H_{\mathsf{IC}}) \cup \{\mathsf{NEW}\}$.

Example 31 Procedure `insertSome` in Figure 5-3 is similar to procedure `insert` in Figure 5-2, except that the node inserted is created inside the procedure. It is therefore referred to in effects via generic summary node NEW. △

We represent all may write effects as a set $\mathsf{mayWr}(\mathsf{proc})$ of triples $\langle n_j, f, n'_j \rangle$ where $n, n'_j \in \mathsf{nodes}_0(H_{\mathsf{IC}})$ and $f \in F$. We represent must write effects as a sequence $\mathsf{mustWr}_j(\mathsf{proc})$ of subsets of the set $K_{\mathsf{IC}}^{-1}(i) \times F \times \mathsf{nodes}_0(H_{\mathsf{IC}})$. Here $1 \leq j \leq \mathsf{mustWrNo}(\mathsf{proc})$.

To simplify the interpretation of the declared procedure effects in terms of concrete reads and writes, we require the union $\cup_i \mathsf{mustWr}_i(\mathsf{proc})$ to be disjoint from the set $\mathsf{mayWr}(\mathsf{proc})$. We also require the nodes $n_1, \ldots, n_k$ in a must write effect $n_1 | \cdots | n_k.f = e_2$ to be individual nodes. This allows strong updates when instantiating effects (Section 5.3.2).

5.1.3 Semantics of Procedure Effects

We now give precise meaning to procedure effects. Our definition is slightly complicated by the desire to capture the set of nodes that are actually read in an execution while still allowing a certain amount of observational equivalence for write effects.

The effects of procedure proc define a subset of permissible program traces in the following way. Consider a concrete heap H_c with role assignment ρ_c such that

```
procedure insertSome(l : L)
nodes ln;
edges l-> ln,
      ln -next-> LN|null;
effects ln|LN . next = NEW,
        NEW.next = LN|null;
aux c, p, x;
{
  p = l;
  c = l.next;
  while (c!=null) {
    p = c;
    c = p.next;
  }
  x = new;
  p.next = x;
  x.next = c;
  setRole(x:LN);
}
```

Figure 5-3: Insert Procedure with Object Allocation

$\langle H_c, \rho_c \rangle \, \alpha_0 \langle H_{\mathsf{lc}}, \rho_{\mathsf{lc}}, K_{\mathsf{lc}} \rangle$ with graph homomorphism h_0 from Definition 28. Consider a trace T starting from a state with heap H_c and role assignment ρ_c. Extract the subsequence of all loads and stores in trace T. Replace Load `x=y.f` by concrete read `read` o_x where o_x is the concrete object referenced by `x` at the point of Load, and replace Store `x.f=y` by a concrete write $o_x.f = o_y$ where o_x is the object referenced by `x` and o_y object referenced by `y` at the point of Store. Let $p_1, \ldots, p_k$ be the sequence of all concrete read statements and $q_1, \ldots, q_k$ the sequence of all concrete write statements. We say that trace T starting at H_c conforms to the effects iff for all choices of h_0 the following conditions hold:

1. $h_0(o) \in \mathsf{read}(\mathsf{proc})$ for every p_i of the form `read` o
2. there exists a subsequence $q_{i_1}, \ldots, q_{i_t}$ of $q_1, \ldots, q_k$ such that
   (a) executing $q_{i_1}, \ldots, q_{i_t}$ on H_c yields the same result as executing the entire sequence $q_1, \ldots, q_k$
   (b) the sequence $q_{i_1}, \ldots, q_{i_t}$ implements write effects of procedure proc

A typical way to obtain a sequence $q_{i_1}, \ldots, q_{i_t}$ from the sequence $q_1, \ldots, q_k$ is to consider only the last write for each pair $\langle o_i, f \rangle$ of object and field.

We say that a sequence $q_{i_1}, \ldots, q_{i_t}$ implements write effects $\mathsf{mayWr}(\mathsf{proc})$ and $\mathsf{mustWr}_i(\mathsf{proc})$ for $1 \leq i \leq i_0$, $i_0 = \mathsf{mustWrNo}$ if and only if there exists an injection $s : \{1, \ldots, i_0\} \to \{i_1, \ldots, i_t\}$ such that

1. $\langle h'(o), f, h'(o')\rangle \in \mathsf{mustWr}_i(\mathsf{proc})$ for every concrete write $q_{s(i)}$ of the form $o.f = o'$, and

2. $\langle h'(o), f, h'(o')\rangle \in \mathsf{mayWr}(\mathsf{proc})$ for all concrete writes q_i of the form $o.f = o'$ for $i \in \{i_1, \ldots, i_t\} \setminus \{s(1), \ldots, s(i_0)\}$.

Here $h'(n) = h_0(n)$ for $n \in \mathsf{nodes}(H_c)$ where H_c is the initial concrete heap and $h'(n) = \mathsf{NEW}$ otherwise.

It is possible (although not very common) for a single concrete heap H_c to have multiple homomorphisms h_0 to the initial context H_{IC}. Note that in this case we require the trace T to conform to effects for *all* possible valid choices of h_0. This places the burden of multiple choices of h_0 on procedure transfer relation verification (Section 5.2) but in turn allows the context matching algorithm in Section 5.3.1 to select an arbitrary homomorphism between a caller's role graph and an initial context.

5.2 Verifying Procedure Transfer Relations

In this section we show how the analysis makes sure that a procedure conforms to its specification, expressed as an initial context with a list of effects. To verify procedure effects, we extend the analysis representation from Section 4.1. A non-error role graph is now a tuple $\langle H, \rho, K, \tau, E\rangle$ where:

1. $\tau : \mathsf{nodes}(H) \to \mathsf{nodes}_0(H_{\mathsf{IC}})$ is initial context transformation that assigns an initial context node $\tau(n) \in \mathsf{nodes}(H_{\mathsf{IC}})$ to every node n representing objects that existed prior to the procedure call, and assigns NEW to every node representing objects created during procedure activation;

2. $E \subseteq \cup_i \mathsf{mustWr}_i(\mathsf{proc})$ is a list of must write effects that procedure has performed so far.

The initial context transformation τ tracks how objects have moved since the beginning of procedure activation and is essential for verifying procedure effects which refer to initial context nodes.

We represent the list E of performed must effects as a partial map from the set $K_{\mathsf{IC}}^{-1}(i) \times F$ to $\mathsf{nodes}_0(H_{\mathsf{IC}})$. This allows the analysis to perform must effect folding by recording only the last must effect for every pair $\langle n, f\rangle$ of individual node n and field f.

5.2.1 Role Graphs at Procedure Entry

Our role analysis creates the set of role graphs at procedure entry point from the initial context $\mathsf{context}(\mathsf{proc})$. This is simple because role graphs and the initial context have similar abstraction relations (Sections 4.1 and 5.1). The difference is that parameters in role graphs point to exactly one node, and parameter nodes are onstage nodes in role graphs which means that all their edges are "must" edges.

Figure 5-4 shows the construction of the initial set of role graphs. First the graph H_0 is created such that every parameter $\mathsf{param}_i(\mathsf{proc})$ references exactly one

$$
\begin{aligned}
[\![\texttt{entry}\bullet]\!] = \Big\{ \langle H, \rho, K, \tau, E\rangle \;\Big|\; & \\
& P : \{\mathsf{proc}\} \times \{\mathsf{param}_i(\mathsf{proc})\}_i \rightarrow N, P \subseteq H_{\mathsf{IC}} \\
& H_0 = (H_{\mathsf{IC}} \setminus \{\mathsf{proc}\} \times \mathsf{param}(\mathsf{proc}) \times N) \cup P \\
& n_i = P(\mathsf{proc}, \mathsf{param}_i(\mathsf{proc})) \\
& H_1 \subseteq H_0 \\
& H_1 \setminus H_0 \subseteq \{\langle n', f, n''\rangle \mid \{n_1, n_2\} \cap \{n_i\}_i \neq \emptyset\} \\
& \forall j : \mathsf{localCheck}(n_j, \langle H, \rho, K\rangle, \mathsf{nodes}(H_1)) \\
& H_1 \overset{n_1}{\parallel} H_2 \overset{n_2}{\parallel} \cdots \overset{n_p}{\parallel} H \\
& \rho = \rho_{\mathsf{IC}} \\
& K = K_{\mathsf{IC}} \\
& \tau = \rho_{\mathsf{IC}} \\
& E = \emptyset \Big\}
\end{aligned}
$$

Figure 5-4: The Set of Role Graphs at Procedure Entry

parameter node n_i. Next graph H_1 is created by using $\mathsf{localCheck}$ to ensure that parameter nodes have the appropriate number of edges. Finally, the instantiation is performed on parameter nodes to ensure acyclicity constraints if the initial context does not make them explicit already.

Statement s	Transition	Constraints
`x = y.f`	$\langle H \uplus \{\mathsf{proc}, \mathsf{x}, n_x\}, \rho, K, \tau, E\rangle \overset{\mathsf{st}}{\Longrightarrow} \langle H \uplus \{\mathsf{proc}, \mathsf{x}, n_f\}, \rho, K, \tau, E\rangle$	$\langle \mathsf{proc}, \mathsf{y}, n_y\rangle, \langle n_y, f, n_f\rangle \in H$ $\tau(n_f) \in \mathsf{read}(\mathsf{proc})$
`x = y.f`	$\langle H \uplus \{\mathsf{proc}, \mathsf{x}, n_x\}, \rho, K, \tau, E\rangle \overset{\mathsf{st}}{\Longrightarrow} \perp_G$	$\langle \mathsf{proc}, \mathsf{y}, n_y\rangle, \langle n_y, f, n_f\rangle \in H$ $\tau(n_f) \notin \mathsf{read}(\mathsf{proc})$
`x.f = y`	$\langle H \uplus \{n_x, f, n_f\}, \rho, K, \tau, E\rangle \overset{\mathsf{st}}{\Longrightarrow} \langle H \uplus \{n_x, f, n_y\}, \rho, K, \tau, E\rangle$	$\langle \mathsf{proc}, \mathsf{x}, n_x\rangle, \langle \mathsf{proc}, \mathsf{y}, n_y\rangle \in H$ $\langle \tau(n_x), f, \tau(n_y)\rangle \in \mathsf{mayWr}(\mathsf{proc})$
`x.f = y`	$\langle H \uplus \{n_x, f, n_f\}, \rho, K, \tau, E\rangle \overset{\mathsf{st}}{\Longrightarrow} \langle H \uplus \{n_x, f, n_y\}, \rho, K, \tau, E'\rangle$	$\langle \mathsf{proc}, \mathsf{x}, n_x\rangle, \langle \mathsf{proc}, \mathsf{y}, n_y\rangle \in H$ $\langle \tau(n_x), f, \tau(n_y)\rangle \in \cup_i \mathsf{mustWr}_i(\mathsf{proc})$ $E' = \mathsf{updateWr}(E, \langle \tau(n_x), f, \tau(n_y)\rangle)$
`x.f = y`	$\langle H \uplus \{n_x, f, n_f\}, \rho, K, \tau, E\rangle \overset{\mathsf{st}}{\Longrightarrow} \perp_G$	$\langle \mathsf{proc}, \mathsf{x}, n_x\rangle, \langle \mathsf{proc}, \mathsf{y}, n_y\rangle \in H$ $\langle \tau(n_x), f, \tau(n_y)\rangle \notin \mathsf{mayWr}(\mathsf{proc}) \cup$ $\cup_i \mathsf{mustWr}_i(\mathsf{proc})$
`x = new`	$\langle H \uplus \{\mathsf{proc}, \mathsf{x}, n_x\}, \rho, K, \tau, E\rangle \overset{\mathsf{st}}{\Longrightarrow} \langle H \uplus \{\mathsf{proc}, \mathsf{x}, n_n\}, \rho, K, \tau', E\rangle$	n_n fresh $\tau' = \tau[n_n \mapsto \mathsf{NEW}]$

$$\mathsf{updateWr}(E, \langle n_1, f, n_2\rangle) = E[\langle n_1, f\rangle \mapsto n_2]$$

Figure 5-5: Verifying Load, Store, and New Statements

5.2.2 Verifying Basic Statements

To ensure that a procedure conforms to its transfer relation the analysis uses the initial context transformation τ to assign every Load and Store statement to a declared effect. Figure 5-5 shows new symbolic execution of Load, Store and New statements.

The symbolic execution of Load statement `x=y.f` makes sure that the node being loaded is recorded in some read effect. If this is not the case, an error is reported.

The symbolic execution of the Store statement `x.f=y` first retrieves nodes $\tau(n_x)$ and $\tau(n_y)$ in the initial role graph $\mathsf{context}$ that correspond to nodes n_x and n_y in the current role graph. If the effect $\langle\tau(n_x), f, \tau(n_y)\rangle$ is declared as a may write effect the execution proceeds as usual. Otherwise, the effect is used to update the list E of must-write effects. The list E is checked at the end of procedure execution.

The symbolic execution of the New statement updates the initial context transformation τ assigning $\tau(n_n) = \mathsf{NEW}$ for the new node n_n.

The τ transformation is similarly updated during other abstract heap operations. Instantiation of node n' into node n_0 assigns $\tau(n_0) = \tau(n')$, split copies values of τ into the new set of isomorphic nodes, and normalization does not merge nodes n_1 and n_2 if $\tau(n_1) \neq \tau(n_2)$.

5.2.3 Verifying Procedure Postconditions

At the end of the procedure, the analysis verifies that $\rho(n_i) = \mathsf{postR}_i(\mathsf{proc})$ where $\langle\mathsf{proc}, \mathsf{param}_i(\mathsf{proc}), n_i\rangle \in H$, and then performs node check on all onstage nodes using predicate $\mathsf{nodeCheck}(n, \langle H, \rho, K\rangle, \mathsf{nodes}(H))$ for all $n \in \mathsf{onstage}(H)$.

At the end of the procedure, the analysis also verifies that every performed effect in $E = \{e_1, \ldots, e_k\}$ can be attributed to exactly one declared must effect. This means that $k = \mathsf{mustWrNo}(\mathsf{proc})$ and there exists a permutation s of set $\{1, \ldots, k\}$ such that $e_{s(i)} \in \mathsf{mustWr}_i(\mathsf{proc})$ for all i.

5.3 Analyzing Call Sites

The set of role graphs at the procedure call site is updated based on the procedure transfer relation as follows. Consider procedure proc containing call site $p \in N_{\mathsf{CFG}}(\mathsf{proc})$ with procedure call $\mathsf{proc}'(x_1, \ldots, x_p)$. Let $\langle H_{\mathsf{IC}}, \rho_{\mathsf{IC}}, K_{\mathsf{IC}}\rangle = \mathsf{context}(\mathsf{proc}')$ be the initial context of the callee.

Figure 5-6 shows the transfer function for procedure call sites. It has the following phases:

1. **Parameter Check** ensures that roles of parameters conform to the roles expected by the callee proc'.
2. **Context Matching** ($\mathsf{matchContext}$) ensures that the caller's role graphs represent a subset of concrete heaps represented by $\mathsf{context}(\mathsf{proc}')$. This is done by deriving a mapping μ from the caller's role graph to $\mathsf{nodes}(H_{\mathsf{IC}})$.
3. **Effect Instantiation** ($\xrightarrow{\mathsf{FX}}$) uses effects $\mathsf{mayWr}(\mathsf{proc}')$ and $\mathsf{mustWr}_i(\mathsf{proc}')$ in order to approximate all structural changes to the role graph that proc' may perform.
4. **Role Reconstruction** ($\xrightarrow{\mathsf{RR}}$) uses final roles for parameter nodes and global role declarations $\mathsf{postR}_i(\mathsf{proc}')$ to reconstruct roles of all nodes in the part of the role graph representing modified region of the heap.

$$
\begin{aligned}
&[\![\mathsf{proc}'(x_1,\ldots,x_p)]\!](\mathcal{G}) = \\
&\quad \mathsf{if}\ \exists G \in \mathcal{G} : \neg\mathsf{paramCheck}(G)\ \mathsf{then}\ \{\bot_G\} \\
&\quad\ \ \mathsf{else\ try}\ \mathcal{G}_1 = \mathsf{matchContext}(\mathcal{G}) \\
&\qquad\quad \mathsf{if\ failed\ then}\ \{\bot_G\} \\
&\qquad\quad\ \ \mathsf{else}\ \{G'' \mid \langle G,\mu\rangle \in \mathcal{G}_1 \\
&\qquad\qquad\qquad \langle \mathsf{addNEW}(G),\mu\rangle \xrightarrow{\mathsf{FX}} \langle G',\mu\rangle \xrightarrow{\mathsf{RR}} G''\}
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{paramCheck}(\langle H,\rho,K,\tau,E\rangle)\ \text{iff} \\
&\quad \forall n_i : \mathsf{nodeCheck}(n_i, G, \mathsf{offstage}(H) \cup \{n_i\}_i) \\
&\quad n_i \text{ are such that } \langle \mathsf{proc}, x_i, n_i\rangle \in H
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{addNEW}(\langle H,\rho,K,\tau,E\rangle) = \\
&\quad \langle H \cup \{n_0\} \times F \times \{\mathsf{null}\}, \\
&\quad \rho[n_0 \mapsto \mathsf{unknown}], \\
&\quad K[n_0 \mapsto s], \\
&\quad \tau[n_0 \mapsto \mathsf{NEW}], \\
&\quad E\rangle \\
&\text{where } n_0 \text{ is fresh in } H
\end{aligned}
$$

Figure 5-6: Procedure Call

The parameter check requires $\mathsf{nodeCheck}(n_i, G, \mathsf{offstage}(H) \cup \{n_i\}_i)$ for the parameter nodes n_i. The other three phases are explained in more detail below.

5.3.1 Context Matching

Figure 5-7 shows our context matching function. The matchContext function takes a set $\mathcal{G}$ of role graphs and produces a set of pairs $\langle G,\mu\rangle$ where $G = \langle H,\rho,K,\tau,E\rangle$ is a role graph and μ is a homomorphism from H to H_{IC}. The homomorphism μ guarantees that $\alpha^{-1}(G) \subseteq \alpha_0^{-1}(\mathsf{context}(\mathsf{proc}'))$ since the homomorphism h_0 from Definition 28 can be constructed from homomorphism h in Definition 22 by putting $h_0 = \mu \circ h$. This implies that it is legal to call proc' with any concrete graph represented by G.

The algorithm in Figure 5-7 starts with empty maps $\mu = \mathsf{nodes}(G) \times \{\bot\}$ and extends μ until it is defined on all $\mathsf{nodes}(G)$ or there is no way to extend it further. It proceeds by choosing a role graph $\langle H,\rho,K,\tau,E\rangle$ and node n_0 for which the mapping μ is not defined yet. It then finds candidates in the initial context that n_0 can be mapped to. The candidates are chosen to make sure that μ remains a homomorphism. The accessibility requirement—that a procedure may see no nodes with incorrect role—is enforced by making sure that nodes in inaccessible are never mapped into nodes in read for the callee. As long as this requirement holds, nodes in inaccessible can be mapped onto nodes of any role since their role need not be correct anyway. We generally require that the set $\mu^{-1}(n_0')$ for individual node n_0' in the initial context contain at most one node, and this node must be individual. In contrast, there might

$$\mathsf{matchContext}(\mathcal{G}) = \mathsf{match}(\{\langle G, \mathsf{nodes}(G) \times \{\bot\}\rangle \mid G \in \mathcal{G}\})$$

$$\mathsf{match} : \mathcal{P}(\mathsf{RoleGraphs} \times (N \cup \{\bot\})^N) \rightharpoonup \mathcal{P}(\mathsf{RoleGraphs} \times N^N)$$

$$
\begin{array}{l}
\mathsf{match}(\Gamma) = \\
\;\Gamma_0 := \{\langle G, \mu\rangle \in \Gamma \mid \mu^{-1}(\bot) \neq \emptyset\}; \\
\;\mathsf{if}\ \Gamma_0 = \emptyset\ \mathsf{then}\ \ \mathsf{return}\ \Gamma; \\
\;\langle\langle H, \rho, K, \tau, E\rangle, \mu\rangle :=\ \mathsf{choose}\ \Gamma_0; \\
\;\Gamma' = \Gamma \setminus \langle\langle H, \rho, K, \tau, E\rangle, \mu\rangle; \\
\;\mathsf{paramnodes} := \{n \mid \exists i : \langle \mathsf{proc}, x_i, n\rangle \in H\}; \\
\;\mathsf{inaccessible} := \mathsf{onstage}(H) \setminus \mathsf{paramnodes}; \\
\;n_0 :=\ \mathsf{choose}\ \mu^{-1}(\bot); \\
\;\mathsf{candidates} := \{n' \in \mathsf{nodes}(H_{\mathsf{IC}}) \mid \\
\qquad\qquad (n_0 \notin \mathsf{inaccessible}\ \mathrm{and}\ \rho_{\mathsf{IC}}(n') = \rho(n_0))\ \mathrm{or} \\
\qquad\qquad (n_0 \in \mathsf{inaccessible}\ \mathrm{and}\ n' \notin \mathsf{read}(\mathsf{proc}'))\} \\
\qquad \displaystyle\bigcap_{\substack{\langle n_0, f, n\rangle \in H \\ \mu(n) \neq \bot}} \left\{n' \,\middle|\, \langle n', f, \mu(n)\rangle \in H_{\mathsf{IC}}\right\} \\
\qquad \displaystyle\bigcap_{\substack{\langle n, f, n_0\rangle \in H \\ \mu(n) \neq \bot}} \left\{n' \,\middle|\, \langle \mu(n), f, n'\rangle \in H_{\mathsf{IC}}\right\}; \\
\;\mathsf{if}\ \mathsf{candidates} = \emptyset\ \mathsf{then}\ \ \mathsf{fail}\ ; \\
\;\mathsf{if}\ \mathsf{candidates} = \{n'_0\}, K(n_0) = s, K_{\mathsf{IC}}(n'_0) = i, \mu^{-1}(n'_0) = \emptyset \\
\quad\ \mathsf{then}\ \mathsf{match}(\Gamma' \cup \{\langle G', \mu[n_1 \mapsto n'_0]\rangle \mid \langle H, \rho, K, \tau, E\rangle \overset{n_1}{\underset{n_0}{\Uparrow}} G'\}) \\
\;\;\mathsf{else}\ n'_0 :=\ \mathsf{choose}\ \{n' \in \mathsf{candidates} \mid K(n') = s\ \mathrm{or} \\
\qquad\qquad\qquad\qquad (K(n_0) = i, \mu^{-1}(n') = \emptyset)\} \\
\qquad\ \mathsf{match}(\Gamma' \cup \langle\langle H, \rho, K, \tau, E\rangle, \mu[n_0 \mapsto n'_0]\rangle);
\end{array}
$$

Figure 5-7: The Context Matching Algorithm be many individual and summary nodes mapped onto a summary node. We relax this requirement by performing instantiation of a summary node of the caller if, at some point, that is the only way to extend the mapping μ (this corresponds to the first recursive call in the definition of match in Figure 5-7).

The algorithm is nondeterministic in the order in which nodes to be matched are selected. One possible ordering of nodes is depth-first order in the role graph starting from parameter nodes. If some nondeterministic branch does not succeed, the algorithm backtracks. The function fails if all branches fail. In that case the procedure call is considered illegal and $\perp_G$ is returned. The algorithm terminates since every procedure call lexicographically increases the sorted list of numbers $|\mu[\mathsf{nodes}(H)]|$ for $\langle\langle H, \rho, K, \tau, E\rangle, \mu\rangle \in \Gamma$.

5.3.2 Effect Instantiation

The result of the matching algorithm is a set of pairs $\langle G, \mu\rangle$ of role graphs and mappings. These pairs are used to instantiate procedure effects in each of the role graphs of the caller. Figure 5-8 gives rules for effect instantiation. The analysis first verifies that the region read by the callee is included in the region read by the caller. Then it uses map μ to find the inverse image S of the performed effects. The effects in S are grouped by the source n and field f. Each field $n.f$ is applied in sequence. There are three cases when applying an effect to $n.f$:

1. There is only one node target of the write in $\mathsf{nodes}(H)$ and the effect is a must write effect. In this case we do a strong update.
2. The condition in 1) is not satisfied, and the node n is offstage. In this case we conservatively add all relevant edges from S to H.
3. The condition in 1) is not satisfied, but the node n is onstage i.e. it is a parameter node[1]. In this case there is no unique target for $n.f$, and we cannot add multiple edges either as this would violate the invariant for onstage nodes. We therefore do case analysis choosing which effect was performed last. If there are no must effects that affect n, then we also consider the case where the original graph is unchanged.

5.3.3 Role Reconstruction

Procedure effects approximate structural changes to the heap, but do not provide information about role changes for non-parameter nodes. We use the role reconstruction algorithm $\xrightarrow{\mathsf{RR}}$ in Figure 5-9 to conservatively infer possible roles of nodes after the procedure call based on role changes for parameters and global role definitions.

Role reconstruction first finds the set N_0 of all nodes that might be accessed by the callee since these nodes might have their roles changed. Then it splits each node

[1]Non-parameter onstage nodes are never affected by effects, as guaranteed by the matching algorithm.

$$\langle\langle H,\rho,K,\tau,E\rangle,\mu\rangle \xrightarrow{\mathsf{FX}} \langle\bot_G,\mu\rangle \text{ where } \quad \tau[\mu^{-1}[\mathsf{read}(\mathsf{proc}')]] \not\subseteq \mathsf{read}(\mathsf{proc})$$

$$\langle\langle H,\rho,K,\tau,E\rangle,\mu\rangle \xrightarrow{\mathsf{FX}} G_t \text{ where } \quad \tau[\mu^{-1}[\mathsf{read}(\mathsf{proc}')]] \subseteq \mathsf{read}(\mathsf{proc})$$

$$\langle H,\rho,K,\tau,E\rangle \overset{n_1,f_1}{\vdash} G_1 \vdash \cdots \overset{n_t,f_t}{\vdash} G_t$$

$$S = \{\langle n,f,n'\rangle \in H \mid \langle\mu(n),f,\mu(n')\rangle \in \mathsf{mayWr}(\mathsf{proc}') \cup \cup_i \mathsf{mustWr}_i(\mathsf{proc}')\}$$

$$\{\langle n_1,f_1\rangle,\ldots,\langle n_t,f_t\rangle\} = \{\langle n,f\rangle \mid \langle n,f,n'\rangle \in S\}$$

Single Write Effect Instantiation:

$$\langle H_1,\rho_1,K_1,\tau_1,E_1\rangle \overset{n,f}{\vdash} G'$$

iff

case	condition	result
deterministic effect	$\{n_1 \mid \langle n,f,n_1\rangle \in S\} = \{n_0\}$ and $\exists i : \langle\mu(n),f,\mu(n_0)\rangle \in \mathsf{mustWr}_i(\mathsf{proc}')$	$G' = \langle H_2,\rho_1,K_1,\tau_1,E_2\rangle$ $H_2 = H_1 \setminus \{\langle n,f,n_1\rangle \mid \langle n,f,n_1\rangle \in H_1\}$ $\cup\{\langle n,f,n_0\rangle\}$ $E_2 = \mathsf{updateWr}(E_1,\langle\tau(n),f,\tau(n_0)\rangle)$
nondeterministic effect for non-parameters	$\|\{n_1 \mid \langle n,f,n_1\rangle \in S\}\| > 1$ or $\exists n_1 : \langle\mu(n),f,\mu(n_1)\rangle \in \mathsf{mayWr}(\mathsf{proc}')$ $n \in \mathsf{offstage}(H)$ $\{\langle\tau(n),f,\tau(n_1)\rangle \mid \langle n,f,n_1\rangle \in S\} \subseteq \mathsf{mayWr}(\mathsf{proc})$	$G' = \langle H_2,\rho_1,K_1,\tau_1,E_2\rangle$ $H_2 = \mathsf{orem}(H_1)\cup$ $\{\langle n,f,n_1\rangle \mid \langle n,f,n_1\rangle \in S\}$
	$\|\{n_1 \mid \langle n,f,n_1\rangle \in S\}\| > 1$ or $\exists n_1 : \langle\mu(n),f,\mu(n_1)\rangle \in \mathsf{mayWr}(\mathsf{proc}')$ $n \in \mathsf{offstage}(H)$ $\{\langle\tau(n),f,\tau(n_1)\rangle \mid \langle n,f,n_1\rangle \in S\} \not\subseteq \mathsf{mayWr}(\mathsf{proc})$	$G' = \bot_G$
nondeterministic effect for parameters	$\|\{n_1 \mid \langle n,f,n_1\rangle \in S\}\| > 1$ or $\exists n_1 : \langle\mu(n),f,\mu(n_1)\rangle \in \mathsf{mayWr}(\mathsf{proc}')$ $n \notin \mathsf{offstage}(H)$ $\{\langle\tau(n),f,\tau(n_1)\rangle \mid \langle n,f,n_1\rangle \in S\} \subseteq \mathsf{mayWr}(\mathsf{proc})$	$G' = \langle H_2,\rho_1,K_1,\tau_1,E_2\rangle$ $H_0 = H_1 \setminus \{\langle n,f,n_1\rangle \mid \langle n,f,n_1\rangle \in H_1\}$ $H_2 = H_1$ or $H_2 = H_0 \cup \{\langle n,f,n_1\rangle\}$ $\langle n,f,n_1\rangle \in S$
	$\neg(\{n_1 \mid \langle n,f,n_1\rangle \in S\} = \{n_1\}$ and $\exists i : \langle\mu(n),f,\mu(n_0)\rangle \in \mathsf{mustWr}_i(\mathsf{proc}'))$ $n \notin \mathsf{offstage}(H)$ $\{\langle\tau(n),f,\tau(n_1)\rangle \mid \langle n,f,n_1\rangle \in S\} \not\subseteq \mathsf{mayWr}(\mathsf{proc})$	$G' = \bot_G$

$$\mathsf{orem}(H_1) = \begin{cases} H_1 \setminus \{\langle n,f,n'\rangle \mid \langle n,f,n'\rangle \in H_1\}, & \text{if } \exists i\, \exists n' : \langle\mu(n),f,\mu(n')\rangle \in \mathsf{mustWr}_i(\mathsf{proc}') \\ H_1, & \text{otherwise} \end{cases}$$

Figure 5-8: Effect Instantiation

$$\langle\langle H,\rho,K,\tau,E\rangle,\mu\rangle \xrightarrow{\mathsf{RR}} \langle H',\rho',K',\tau',E'\rangle$$

$$\begin{array}{l}
\langle \mathsf{proc}, x_i, n_i\rangle \in H \\
N_0 = \mu^{-1}[\mathsf{read}(\mathsf{proc}')] \\
s : N_0 \times R \to N \text{ where } s(n,r) \text{ are all different nodes fresh in } H \\
\rho' = \rho \setminus (N_0 \times R) \cup \{\langle s(n,r), r\rangle \mid n \in N_0, r \in R\} \\
\qquad\quad \setminus(\{n_i\}_i \times R) \cup \{\langle n_i, \mathsf{postR}_i(\mathsf{proc})\rangle\} \\
K'(s(n,r)) = K(n) \\
\tau'(s(n,r)) = \tau(n) \\
E' = E \\
H_0 = H \setminus \{\langle n_1, f, n_2\rangle \mid n_1 \in N_0 \text{ or } n_2 \in N_0\} \\
\qquad\quad \cup \{\langle s(n_1,r_1), f, s(n_2,r_2)\rangle \mid \langle n_1,f,n_2\rangle \in H, \langle r_1,f,r_2\rangle \in \mathsf{RRD}\} \\
\qquad\quad \cup \{\langle n_1, f, s(n_2,r_2)\rangle \mid \langle n_1,f,n_2\rangle \in H, \langle \rho_{\mathsf{lc}}(\mu(n_1)),f,r_2\rangle \in \mathsf{RRD}\} \\
\qquad\quad \cup \{\langle s(n_1,r_1), f, n_2\rangle \mid \langle n_1,f,n_2\rangle \in H, \langle r_1,f,\rho_{\mathsf{lc}}(\mu(n_2))\rangle \in \mathsf{RRD}\} \\
H' = \mathsf{GC}(H_0)
\end{array}$$

Figure 5-9: Call Site Role Reconstruction

$n \in N_0$ into $|R|$ different nodes $\rho(n,r)$, one for each role $r \in R$. The node $\rho(n,r)$ represents the subset of objects that were initially represented by n and have role r after procedure executes. The edges between nodes in the new graph are derived by simultaneously satisfying 1) structural constraints between nodes of the original graph; and 2) global role constraints from the role reference diagram. The nodes $\rho(n,r)$ not connected to the parameter nodes are garbage collected in the role graph. In practice, we generate nodes $\rho(n,r)$ and edges on demand starting from parameters making sure that they are reachable and satisfy both kinds of constraints.

Chapter 6

Extensions

This chapter presents extensions of the basic role system. The *multislot* extension allows statically unbounded number of aliases for objects. *Root variables* allow stack frames to be treated as the source of aliases in role definitions. *Singleton roles* allow role declarations to specify that there is only one object of a given role. The extension for *cascading role changes* allows the analysis to verify more complex role changes. The extension to *partial roles* allows mutually independent role properties to be specified separately and then combined.

6.1 Multislots

A multislot $\langle r', f \rangle \in \mathsf{multislots}(r)$ in the definition of role r allows any number of aliases $\langle o', f, o \rangle \in H_c$ for $\rho_c(o') = r'$ and $\rho_c(o) = r$. We require multislots $\mathsf{multislots}(r)$ to be disjoint from all $\mathsf{slot}_i(r)$. To handle multislots in role analysis we relax the condition 5) in Definition 22 of the abstraction relation by allowing h to map more than one concrete edge $\langle o', f, o \rangle$ onto abstract edge $\langle n', f, n \rangle \in H$ terminating at an onstage node n provided that $\langle \rho(n'), f \rangle \in \mathsf{multislots}(\rho(n))$. The $\mathsf{nodeCheck}$ and expansion relation $\preceq$ are then extended appropriately. Note that a role graph does not represent the exact number of references that fill each multislot. The analysis therefore does not attempt to recognize actions that remove the last reference from the multislot. Once an object plays a role with a multislot, all subsequent roles that it plays must also have the multislot.

6.2 Root Variables

Root variables allow roles to be defined not only by heap references from other nodes but also by references from procedure variables. The root variables are treated like heap references for the purpose of role consistency; they are references from stack frame objects. A procedure with root variables induces a role with fields corresponding to root variables and no slots.

Example 32 Let us reconsider the scheduler example in Figure 1-2. We can require the `LiveHeader` node to be referenced by the root variable `processes` in the procedure `main`, and `RunningHeader` to be referenced by the root variable `running` in the following way.

```
role LiveHeader {
  fields first : LiveList | null;
  slots  main.processes;
}
role RunningHeader {
  fields next : RunningProc | RunningHeader,
         prev : RunningProc | RunningHeader;
  slots  main.running,
         RunningHeader.next | RunningProc.next,
         RunningHeader.prev | RunningProc.prev;
  identities next.prev, prev.next;
}

procedure main()
rootvar processes : LiveHeader | null,
        running   : RunningHeader | null;
{ ... }
```

This implicitly generates a role definition for the `main` procedure.

```
role main {
  fields processes : LiveHeader,
         running : RunningHeader;
}
```

$\triangle$

6.3 Singleton Roles

Singleton roles are a simple way to improve the precision of role specifications and role analysis by indicating roles for which there is only a single heap object of that role. Singleton roles are often referred to from root variables.

We say that the predicate $\mathsf{singleton}(r)$ holds for role $r \in R$ if $|\rho_c^{-1}(r)| \leq 1$ for every valid concrete role assignment ρ_c of a heap created by the program. In essence, this predicate allows distinguishing between individual objects and sets of objects in role definitions.

Example 33 The intention of the definition in Figure 6-1 is to specify a circular singly linked list with a header node. However, the specification in Figure 6-1 is too general. For example, the graph in Figure 6-2 satisfies this specification. If we require $\mathsf{singleton}(\mathtt{H})$, then the graph in Figure 6-2 does not satisfy role declarations any more. $\triangle$

```
role H { // header node
  fields next : H | N;
  slots  H.next | N.next;
}
role N { // internal node
  fields next : H | N;
  slots  H.next | N.next;
}
```

Figure 6-1: Roles for Circular List

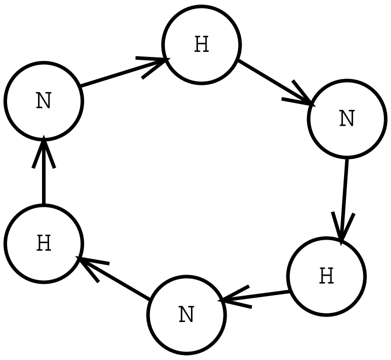


Figure 6-2: An Instance of Role Declarations

The developer can specify values of singleton predicate explicitly. In some cases the analysis alone can infer this information using the following rules:

- procedure activation records are singleton if they are not members of a cycle the call graph;
- if the roles $R_s \in R$ are singleton and $r' \in R$ is such that one of the following criteria holds:
  - there exists $f \in F$ such that $\mathsf{field}_f(r) \subseteq R_s$, or
  - there exists i such that $\mathsf{slot}_i(r') \subseteq R_s$,

  then r' is a singleton role as well.

When analyzing programs with singleton roles, the role analysis maintains the invariant that there is at most one node for each singleton role r by preventing multiple nodes with role r to go offstage. When traversing data structures, the singleton constraint eliminates cases in where two nodes with a singleton role are brought onstage.

```
role BufferNode {
  fields next : BufferNode | null;
  slots  BufferNode.next | main.buffer;
  acyclic next;
}
role WorkNode {
  fields next : WorkNode | null;
         WorkNode.next | main.work;
  acyclic next;
}

procedure main()
rootvar buffer : BufferNode | null,
        work   : WorkNode | null;
auxvar x, y;
{
   // create buffer and work lists
    ...
   // swap buffer and work
   x = buffer;
   y = work;
   buffer = y;
   work = x;
   setRoleCascade(x:WorkNode, y:BufferNode);
}
```

Figure 6-3: Example of a Cascading Role Change

A natural generalization of singleton roles arises in the context of *parametrized roles* [57]. The extension to parametrized roles is orthogonal to the other aspects of roles and we do not consider it in this thesis.

6.4 Cascading Role Changes

In some cases it is desirable to change roles of an entire set of offstage objects without bringing them onstage. We use the statement `setRoleCascade`$(x_1 : r_1, \ldots, x_n : r_n)$ to perform such *cascading role change* of a set of nodes. The need for cascading role changes arises when roles encode reachability properties.

Example 34 Procedure `main` in Figure 6-3 has two root variables, `buffer` and `work`, each being a root for a singly linked acyclic list. Elements of the first list have `BufferNode` role and elements of the second list have `WorkNode` role. At some point procedure swaps the root variables `buffer` and `work`, which requires all nodes in both lists to change the roles. These role changes are triggered by the `setRoleCascade` statement. The statement indicates new roles for onstage nodes, and the analysis cascades role changes to offstage nodes. △

$\langle H, \rho, K, \tau, E\rangle \overset{\texttt{st}}{\rightsquigarrow} \langle H, \rho', K, \tau, E\rangle$ $\mathsf{st} = \texttt{setRoleCascade}(x_1 : r_1, \ldots, x_n : r_n)$	$n_i : \langle \mathsf{proc}, x_i, n_i\rangle \in H$ $\rho'(n_i) = r_i$ $\rho'(n) = \rho(n),\ n \in \mathsf{onstage}(H) \setminus \{n_i\}_i$ $N_0 = \{n \in \mathsf{offstage}(H) \mid \exists n' \in \mathsf{neighbors}(n, H) : \rho(n') \neq \rho'(n')\}$ $\forall n \in N_0 : \mathsf{cascadingOk}(n, H, \rho, K, \rho')$

Figure 6-4: Abstract Execution for `setRoleCascade`

Given a role graph $\langle H, \rho, K, E\rangle$ cascading role change finds a new valid role assignment ρ' where the onstage nodes have desired roles and the roles of offstage nodes are adjusted appropriately. Figure 6-4 shows abstract execution of the `setRoleCascade` statement. Here $\mathsf{neighbors}(n, H)$ denotes nodes in H adjacent to n. The condition $\mathsf{cascadingOk}(n, H, \rho, K, \rho')$ makes sure it is legal to change the role of node n from $\rho(n)$ to $\rho'(n)$ given that the neighbors of n also change role according to ρ'. This check resembles the check for `setRole` statement in Section 4.2.3. Let $r = rho(n)$ and $r' = \rho'(n)$. Then $\mathsf{cascadingOk}(n, H, \rho, K, \rho')$ requires the following conditions:

1. $\langle n, f, n_1\rangle \in H$ implies $\rho'(n_1) \in \mathsf{field}_f(r')$;
2. $\mathsf{slotno}(r') = \mathsf{slotno}(r) = k$, and for every list $\langle n_1, f_1, n\rangle, \ldots, \langle n_k, f_k, n\rangle \in H$ if there is a permutation $p : \{1, \ldots, k\} \to \{1, \ldots, k\}$ such that $\langle \rho(n_i), f_i\rangle \in \mathsf{slot}_{p_i}(r)$, then there is a permutation $p' : \{1, \ldots, k\} \to \{1, \ldots, k\}$ such that $\langle \rho(n_i), f_i\rangle \in \mathsf{slot}_{p_i}(r')$;
3. identity relations were already satisfied or can be explicitly checked: $\langle f, g\rangle \in \mathsf{identities}(\rho'(n))$ implies
   (a) $\langle f, g\rangle \in \mathsf{identities}(\rho(n))$ or
   (b) for all $\langle n, f, n'\rangle \in H$: $K(n') = i$, and if $\langle n', g, n''\rangle \in H$ then $n'' = n$;
4. either $\mathsf{acyclic}(\rho'(n)) \subseteq \mathsf{acyclic}(\rho(n))$ or $\mathsf{acycCheck}(n, \langle H, \rho', K\rangle, \mathsf{offstage}(H))$.

In practice there may be zero or more solutions that satisfy constraints for a given cascading role change. Selecting any solution that satisfies the constraints is sound with respect to the original semantics. A useful heuristic for searching the solution space is to first explore branches with as few roles changed as possible. If no solutions are found, an error is reported.

6.5 Partial Roles

In this section we extend our framework to allow combining roles that specify mutually independent properties of objects. First we generalize field and slot constraints to allow specifying partial information about fields and slots of each role. We then give an alternative semantics of roles where each node is assigned a *set* of roles. A pleasant property of this semantics of roles is that the sets of roles applicable to each field can be defined as the greatest fixpoint of the recursive role definitions. We then sketch an extension of context matching and call site role reconstruction that allows procedures to be analyzed without specifying the full set of roles of objects in the initial role graphs.

6.5.1 Partial Roles and Role Sets

This section introduces *partial roles*. A partial role gives constraints only for a subset of fields and slots. We use the term *simple roles* to refer to non-partial roles considered so far.

Example 35 Consider the definition of a tree in Figure 6-5. This definition specifies

```
role TR { // tree root
  fields left : TN | null,
         right : TN | null;
  left,right slots ;
}
role TN { // tree node
  fields left : TN | null,
         right : TN | null;
  left,right slots : TR.left | TR.right | TN.left | TN.right;
}
```

Figure 6-5: Definition of a Tree

that a data structure is a tree along the `left` and `right` fields, but does not constrain fields other than `left` and `right`. Similarly, the definition of a linked list in Figure 6-6 gives only requirements for the `next` field. Note how definition of `LH` specifies a partial "negative" slot constraint, namely the absence of a next field.

A definition for a threaded tree, for example, can leverage the preceding role definitions to define the composite data structure.

```
role LTN extends TN,LN { // linked tree node
  fields data : Stored;
}
```

Every object playing `LTN` role simultaneously plays `TN` and `LN` roles as well. In general, an object playing more roles satisfies more constraints. △

For partial roles, we change the convention that the fields not mentioned in a `fields` declaration are always constrained to be `null`. Instead, the absence of a

```
role LH { // list header
  fields next : NL | null;
  next slots ;
}
role LN { // list node
  fields next : LN | null;
  next slots LH.next | LN.next;
}
```

Figure 6-6: Definition of a List

field f implies no constraints on the roles that field f references. A slot constraint for a partial role r contains an additional set $\mathsf{scope}(r) = \{f_1, \ldots, f_k\}$ of fields that determine the scope of the slot constraints. A slot declaration gives complete aliases for references along $\mathsf{scope}(r)$ fields, but poses no requirements on aliases from other fields.

Partial role definitions can reuse previous role definitions using the `extends` keyword. We represent the `extends` relationships by the set of roles $\mathsf{subroles}(r)$ for each role r. A set $S \subseteq R$ is *closed* if $\mathsf{subroles}(r) \subseteq S$ for every $r \in S$.

6.5.2 Semantics of Partial Roles

To give the semantics of partial roles we define role-set assignment ρ_c^s to assign a closed set of roles to every object. We say that a role assignment ρ_c is a *choice* of a role-set assignment ρ_c^s iff $\rho_c(r) \in \rho_c^s(r)$ for every role $r \in R$. We first generalize $\mathsf{locallyConsistent}$ to take the role of the object o independently of role assignment ρ_c. This definition is identical to Definition 2 except that the role of the object o is r instead of $\rho_c(o)$.

Definition 36 $\mathsf{locallyConsistent}(o, H_c, \rho_c, r)$ *iff all of the following conditions are met.*

1) *For every field* $f \in F$ *and* $\langle o, f, o' \rangle \in H_c$, $\rho_c(o') \in \mathsf{field}_f(r)$.

2) *Let* $\{\langle o_1, f_1 \rangle, \ldots, \langle o_k, f_k \rangle\} = \{\langle o', f \rangle \mid \langle o', f, o \rangle \in H_c\}$ *be the set of all aliases of node* o. *Then* $k = \mathsf{slotno}(r)$ *and there exists some permutation* p *of the set* $\{1, \ldots, k\}$ *such that* $\langle \rho_c(o_i), f_i \rangle \in \mathsf{slot}_{p_i}(r)$ *for all* i.

3) *If* $\langle o, f, o' \rangle \in H_c$, $\langle o', g, o'' \rangle \in H_c$, *and* $\langle f, g \rangle \in \mathsf{identities}(r)$, *then* $o = o''$.

4) *It is not the case that graph* H_c *contains a cycle* $o_1, f_1, \ldots, o_s, f_s, o_1$ *where* $o_1 = o$ *and* $f_1, \ldots, f_s \in \mathsf{acyclic}(r)$

We now define the local role-set consistency as follows.

Definition 37 $\mathsf{locallyRSConsistent}(o, H_c, \rho_c^s)$ *iff for every* $r \in \rho_c^s(o)$ *there exists a choice* ρ_c *of* ρ_c^s *such that* $\mathsf{locallyConsistent}(o, H_c, \rho_c, r)$*. We say that a heap* H_c *is role-set consistent for a role-set assignment* ρ_c^s *if* $\mathsf{locallyRSConsistent}(o, H_c, \rho_c^s)$ *for every* $o \in \mathsf{nodes}(H_c)$*. We call such role-set assignment* ρ_c^s *a valid role-set assignment.*

We similarly extend the definitions of consistency for a given set of nodes from Definition 20.

The following observations follow from Definition 37:

1. if ρ_c^s is a valid role assignment, then $|\rho_c^s(o)| \geq 1$ for every object o, otherwise there would be no ρ_c which is a choice for ρ_c^s;

2. if $|\rho_c^s(o)| = 1$ for all $o \in \mathsf{nodes}(H_c)$, then heap consistency for partial roles is equivalent to heap consistency for simple roles.

6.5.3 Fixpoint Definition of the Greatest Role Assignment

We first show that the set of all valid role-set assignments has a least upper bound. We first define a partial order on functions from $\mathsf{nodes}(H_c)$ to $\mathcal{P}(R)$.

Definition 38 $\rho_{c1}^s \sqsubseteq \rho_{c2}^s$ *iff* $\rho_{c1}^s(o) \subseteq \rho_{c2}^s(o)$ *for every* $o \in H_c$*.*

We then introduce the pointwise union.

Definition 39

$$(\rho_{c1}^s \sqcup \rho_{c2}^s)(o) = \rho_{c1}^s(o) \cup \rho_{c2}^s(o)$$

The union of two closed role-sets is a closed role-set, so the merge of two role-set assignments is still a role-set assignment. Moreover, if both role-set assignments are valid, the pointwise union is also a valid role-set assignment, as the following property shows.

Property 40 *Let* ρ_{c1}^s *and* ρ_{c2}^s *be valid role-set assignments for the heap* H_c*. Then* $\rho_{c1}^s \sqcup \rho_{c2}^s$ *is also a valid role assignment.*

The property holds because every role assignment ρ_c which is a choice of ρ_{c1}^s or a choice of ρ_{c2}^s is also a choice of $\rho_{c1}^s \sqcup \rho_{c2}^s$.

Because there is a finite number of role-set assignments, Property 40 implies the existence of the *greatest role-set assignment* ρ_c^{sM} which is the merge of all valid role assignments.

Definition 41 *Let* ρ_{c1}^s*, ...,* ρ_{cN}^s *be all valid role assignments for the heap* H_c*. We define the greatest role assignment* ρ_c^{sM} *as*

$$\rho_c^{sM} = \rho_{c1}^s \sqcup \cdots \sqcup \rho_{cN}^s$$

Definition 42 *Let* $\rho_c^s : \mathsf{nodes}(H_c) \to \mathcal{P}(R)$. *Then* $F(\rho_c^s) : \mathsf{nodes}(H_c) \to \mathcal{P}(R)$ *is a defined by*

$$\begin{array}{rcl} F(\rho_c^s)(o) & = & \{r \in \rho_c^s(o) \mid \mathsf{subroles}(r) \subseteq \rho_c^s(o) \textit{ and} \\ & & \textit{there exists a choice } \rho_c \textit{ of } \rho_c^s \textit{ such that} \\ & & \mathsf{locallyConsistent}(o, H_c, \rho_c, r)\} \end{array}$$

Property 43 *The greatest role-set assignment for a concrete heap* H_c *is a greatest fixpoint of function* F.

Proof. It is easy to see that $F(\rho_{c1}^s) \sqsubseteq F(\rho_{c2}^s)$ whenever $\rho_{c1}^s \sqsubseteq \rho_{c2}^s$. Also, $F(\rho_c^s) \sqsubseteq \rho_c^s$ and the empty role-set assignment $\rho_c^s(o) = \emptyset$ is a fixpoint of F.

Let ρ_{c0}^s be such that $\rho_{c0}^s(o) = R$ for all $o \in H_c$. Consider the sequence $F^i(\rho_{c0}^s)$ for $i \geq 0$. There exists i_0 such that $F^i(\rho_{c0}^s) = \rho_{c*}^s$ for $i \geq i_0$ where ρ_{c*}^s is a fixpoint of F. Because $F(\rho_{c*}^s)(o) = \rho_{c*}^s(o)$ for each o, it follows that ρ_{c*}^s is a valid role-set assignment. Moreover, if ρ_c^s is any other valid role-set assignment, then $\rho_c^s \sqsubseteq F^i(\rho_{c0}^s)$ for every i, so $\rho_c^s \sqsubseteq \rho_{c*}^s$. We conclude that the fixpoint ρ_{c*}^s is the greatest valid role assignment ρ_c^{sM}. ∎

6.5.4 Expressibility of Partial Roles

The partial roles allow data structures to be described compositionally. Another nice property of partial roles is that there is a canonical role-set assignment ρ_c^{sM}. A drawback of considering only the greatest role-set assignment is that some data structure constraints are not expressible.

Example 44 The set of cycles of even length can be described using the following simple role definitions.

```
role Even {
  fields next : Odd;
  slots  Odd.next;
}
role Odd {
  fields next : Even;
  slots Even.next;
}
```

No odd length cycle satisfies this role assignment. Each even length cycle $o_1, \ldots, o_{2k}$ has two role assignments ρ_{c1} and ρ_{c2}, where $\rho_{c1}(o_{2i+1}) = \texttt{Odd}$ and $\rho_{c1}(o_{2i}) = \texttt{Even}$, whereas $\rho_{c2}(o_{2i+1}) = \texttt{Even}$ and $\rho_{c2}(o_{2i}) = \texttt{Odd}$.

On the other hand, the same role definitions have unique greatest role assignment $\rho_c^s = \rho_{c1}^s \sqcup \rho_{c2}^s$, where $\rho_c^s(o) = \{\texttt{Even}, \texttt{Odd}\}$ for all o. This role assignment is valid not only for even length cycles, but also for odd length cycles. △

The constraints that can be specified by partial roles and role-set assignments are similar to constraints that can be specified using simple roles and role assignments.

In the absence of acyclicity constraints, given a set of partial role definitions, it is possible to exhibit a set of simple role definitions which capture the same constraints.

This construction introduces a simple role each closed set of partial roles, similar to the construction showing the equivalence of deterministic and nondeterministic finite state automata [61] or deterministic and nondeterministic finite tree automata [34, 15]. Construction is complicated by the form of our slot constraints, but can be done by introducing additional roles that simulate slot constraint conjunction. (The ability to perform conjunction of slot constraints is an easy consequence of the equivalence of slot constraints with the generalized slot constraints in Section A.1.) The construction could also be performed for acyclicity constraints if we generalized them to specify a family of sets of fields and forbid cycles along paths with fields from each of the sets in the family.

Even after performing this construction, it remains the fact that partial roles induce additional partial order structure, which is not available in simple roles.

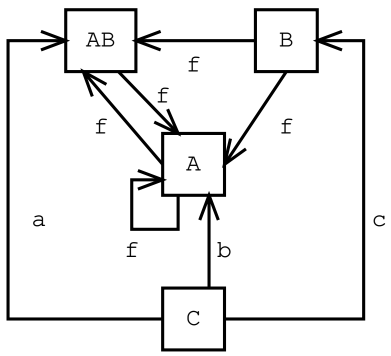
6.5.5 Role Subtyping

We now consider the problem of role subtyping at procedure call sites. A larger set of nodes for a node implies stronger constraints for that node. We would then expect a procedure call to be legal when the caller's role-sets are supersets of role-sets of the initial context. The problem is that a larger set $\rho_c^s(n)$, while implying a stronger constraint on the node n, implies *weaker* constraint on the nodes adjacent to n. The following example shows that the superset conditions on role-sets is in general not sufficient.

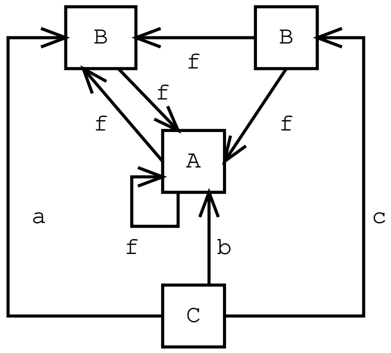
Example 45 Define roles `A` and `B` as follows:

```
role A {
  f slots A.f,
          B.f | A.f;
}
role B { }
role C { }
```

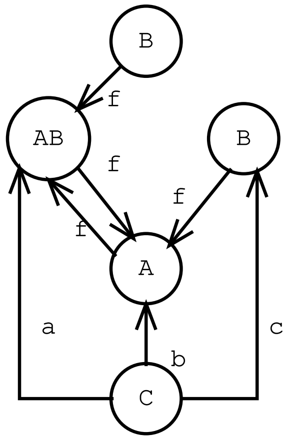
Consider the following role graph in the caller



and assume that the callee has the following initial role graph.



Clearly there is a homomorphism μ from the caller's role graph to the initial role graph such that $\rho^s_1(n) \supseteq \rho^s_2(\mu(n))$ for all nodes n. The following heap is an instance of the caller's role graph.



However, it is not possible to assign sets of roles to objects to make it an instance of the role graph in the initial context. $\triangle$

The following property shows that a simple restriction on slot constraints makes the role-set inclusion criterion valid.

Property 46 *Let* $\langle H, \rho^s, K\rangle$ *and* $\langle H_{\mathsf{IC}}, \rho^s_{\mathsf{IC}}, K_{\mathsf{IC}}\rangle$ *be role graphs and* $\mu : \mathsf{nodes}(H) \to \mathsf{nodes}(H_{\mathsf{IC}})$ *a graph homomorphism such that:*

1. $\rho^s(n) \supseteq \rho^s_{\mathsf{IC}}(\mu(n))$ *for all* $n \in \mathsf{nodes}(H)$*;*

2. *if* $\langle n_1, f, n_0\rangle \in H$*,* $r_0 \in \rho^s_{\mathsf{IC}}(\mu(n_0))$*,* $r_1 \in \rho^s(n_1)$*, and* $\langle r_1, f\rangle \in \mathsf{slot}_i(r_0)$ *for some* i*, then* $\langle r_2, f\rangle \in \mathsf{slot}_i(r_0)$ *for some* $r_2 \in \rho^s_{\mathsf{IC}}(\mu(n_1))$*.*

Let H_c *be a concrete heap such and* ρ^s_{c1} *a valid role-set assignment for* H_c*. Assume that* h *is a homomorphism from* H_c *to* H *such that* $\rho^s_{c1}(o) = \rho^s(h(o))$ *for all* $o \in \mathsf{nodes}(H_c)$*. Define*

$$\rho^s_{c2}(o) = \rho^s_{\mathsf{IC}}(\mu(h(o)))$$

for all $o \in \mathsf{nodes}(H_c)$*. Then* ρ^s_{c2} *is also a valid role-set assignment for* H_c*.*

Proof. To show that ρ^s_{c2} is a valid role-set assignment for H_c, consider any object $o \in \mathsf{nodes}(H_c)$ and one of its roles $r_0 \in \rho^s_{c2}(o)$. Because $r_0 \in \rho^s_{c2}(o)$, identities and acyclicity constraints hold for o. We show that field and slot constraints hold as well.

To show that field constraints of r_0 hold, consider any edge $\langle o, f, o_1\rangle \in H_c$. Then $\langle n, f, n_1\rangle \in H_{\mathsf{IC}}$ where $n = \mu(h(o))$ and $n_1 = \mu(h(o_1))$. Because H_{IC} is a subgraph of the static role diagram, $\mathsf{field}_f(r_0) \cap \rho^s_{\mathsf{IC}}(n_1) \neq \emptyset$, otherwise the edge $\langle n, f, n_1\rangle$ would be superfluous. Since $\rho^s_2(o_1) = \rho^s_{\mathsf{IC}}(n_1)$ by definition of ρ^s_2, we have $\mathsf{field}_f(r_0) \cap \rho^s_2(o_1) \neq \emptyset$ which means that the field constraint for f is satisfied in H_c.

To show that slot constraints of r_0 hold, consider any edge $\langle o_1, f, o\rangle \in H_c$. Because ρ^s_{c1} is a valid role assignment and $r_0 \in \rho^s_{c1}(o)$, there exists slot i and role $r_1 \in \rho^s_{c1}(o_1)$ such that $\langle r_1, f\rangle \in \mathsf{slot}_i(r_0)$. By the assumption 2), since $\langle h(o_1), f, h(o)\rangle \in H$, $r_0 \in \rho^s_{\mathsf{IC}}(h(o))$ and $r_1 \in \rho^s(h(o_1))$, there exists $r_2 \in \rho^s_{\mathsf{IC}}(\mu(h(o_1))$ such that $\langle r_2, f\rangle \in \mathsf{slot}_i(r_0)$. Since $\rho^s_{\mathsf{IC}}(\mu(h(o_1)) = \rho^s_{c2}(o_1)$, it follows that the slot constraint of o is satisfied. ∎

The condition 2) in Property 46 can be replaced by a stronger but simpler condition.

Definition 47 *We say that role r_0* depends *on r_1 iff for some slot i, $\langle r_1, f\rangle \in \mathsf{slot}_i(r_0)$ and there exists another slot $j \neq i$ of role r_0 such that $\langle r_2, f\rangle \in \mathsf{slot}_j(r_0)$ for some role r_2.*

Property 48 *Let $\langle H, \rho^s, K\rangle$ and $\langle H_{\mathsf{IC}}, \rho^s_{\mathsf{IC}}, K_{\mathsf{IC}}\rangle$ be role graphs and $\mu : \mathsf{nodes}(H) \to \mathsf{nodes}(H_{\mathsf{IC}})$ a graph homomorphism such that:*

1') $\rho^s(n) \supseteq \rho^s_{\mathsf{IC}}(\mu(n))$ for all $n \in \mathsf{nodes}(H)$;

2') if $r_1 \in \rho^s(n) \setminus \rho^s_{\mathsf{IC}}(\mu(n))$ for some n, and r_0 depends on r_1, then for all $n' \in \mathsf{nodes}(H_{\mathsf{IC}})$, $r_0 \notin \rho^s_{\mathsf{IC}}(n')$.

Then the condition 2) of Property 46 is satisfied.

Proof. Let $\langle n_1, f, n\rangle \in H$, $r_0 \in \rho^s_{\mathsf{IC}}(n)$, and $r_1 \in \rho^s(H)$ and $\langle r_1, f\rangle \in \mathsf{slot}_i(r_0)$. If $r_1 \in \rho^s_{\mathsf{IC}}(\mu(n))$ then we can take $r_2 = r_1$ and the condition 2) is satisfied. Now assume $r_1 \in \rho^s(n) \setminus \rho^s_{\mathsf{IC}}(\mu(n))$. Since $r_0 \in \rho^s_{\mathsf{IC}}(n)$, by assumption 2'), r_0 does not depend on r_1. This means that i is the only slot of r_0 that contains the field f. Because the edge $\langle \mu(n_1), f, \mu(n)\rangle$ is in H_{IC}, and H_{IC}, it follows that $\langle r_2, f\rangle \in \mathsf{slot}_i(r_0)$ for some $r_2 \in \rho^s_{\mathsf{IC}}(n_1)$. This means that the condition 2) is satisfied. ∎

Based on previous properties we can derive a context matching algorithm that allows role graphs in the call site to have larger sets of roles than nodes in the initial context.

In order to further increase the precision of call site verification, we would like to preserve the larger larger set of role graphs in the caller. This is possible because procedure effects specify which object fields can be modified during execution of the caller. The role reconstruction algorithm for partial roles is similar to algorithm in Figure 5-9 except that it operates on sets of roles instead of individual roles. To consider how to preserve the wider set of roles, consider a role $r \in \rho^s(n) \setminus \rho^s_{\mathsf{IC}}(\mu(n))$. The role reconstruction splits n into a set of nodes each of which has assigned some role-set S. In the absence of write effects the algorithm would need to generate nodes with role-sets S that do not contain r. If the write effects imply that the role r cannot be violated, then only role-sets S containing r need to be generated, which increases the precision and reduces the size of role graphs after the procedure call. To compute the set of roles that are preserved, role reconstruction starts with sets $p(n) = \rho^s(n) \setminus \rho^s_{\mathsf{IC}}(\mu(n))$ assigned to each node n, and iteratively decreases sets $p(n)$ if a $r \in p(n)$ depends on a modified field or previously eliminated role.

We note that, similarly to multislots, partial roles allow a statically unbounded number of aliases. Whereas multislots explicitly give permission for existence of certain aliases, partial roles allow all the existence of aliases not mentioned in the role definition.

Chapter 7

Related Work

In this chapter we present the relationship of our work with previous approaches to program analysis, checking, and verification. We first compare our work with the typestate systems including alias types [82] and calculus of capabilities [19]. We mention the previous work on aliasing control for object-oriented languages [46] and the use of roles in object-oriented modeling [70] and database programming languages [35]. We compare our role analysis with shape types [32], graph types [64], path matrix analysis [36], and parametric shape analysis [78]. We briefly relate our approach to some other interprocedural analyses and examine our work in the context of program verification.

7.1 Typestate Systems

A typestate system for statically verifying initialization properties of values was proposed in [84, 83]. The type state checking was based on a linear two-pass typestate checking algorithm. In this typestate system, the state of an object depends only on its initialization status. This system did not support aliasing of dynamically allocated structures. Aliasing causes problems for typestate-based systems because the declared typestates of all aliases must change whenever the state of the referred object changes. Faced with the complexity of aliasing, [84] resorted to a more controlled language model based on relations. Requiring the relations to exist only between fully initialized objects enables verification of initialization status of objects in the presence of dynamically growing structures. However, this solution is entirely inadequate for the properties which our role system verifies. Our goal is to verify application-specific properties of objects, and not object initialization. Different objects stored in dynamically growing data structures have different application-specific properties, which our system captures as different roles. When object's properties change, our system verifies that the change is consistent with all relations in which the object participates. Our technique is applicable regardless of whether the relations between objects are implemented as pointer fields of records or in some other way. The data-flow analysis [76] performs verification of constraints on relations and sets that implement dynamic structures, but it does not perform instantiation operation like [78] and our role analysis, which leads to the loss of precision when analyzing destructive updates to data structures.

More recently proposed typestate approaches [20, 88, 82, 19] use linear types to support state changes of dynamically allocated objects. The goal of these systems is to enforce safety properties of low-level code, in particular memory management. This is in contrast with our system which aims at verifying higher-level constraints in a language with a garbage collected heap memory model. The capability calculus [19] allows tracking the aliasing of memory regions by doing a form of compile-time reference counting, but does not track aliasing properties of individual objects. Alias types [82] represent precisely the aliasing of individual objects referenced by local variables, but do not support recursive data structures. Recursive alias types [88] allow specification of recursive data structures as unfolding of basic elaboration steps. This allows descriptions of tree-like data structures with parent pointers, but does not permit approximating arbitrary data structures. This property of recursive alias types is shared with shape types [32] and graph types [54] discussed below. Another difference compared to our work is that these type systems present only a *type checking*, and not a *type inference* algorithm, whereas our analysis performs role inference inside each procedure. The application of these type systems to an imperative programming language Vault is presented in [20]. Because it is based on alias types and capability calculus, Vault's type system cannot approximate arbitrary data structures. The type system of Vault tracks run-time resources using unique *keys*. To simplify the type checking, Vault requires the equality of sets of keys at each program point. This is in contrast to predicative data-flow analyses such as role analysis, which track the sets of possible aliasing relationships at each program point. Our approach makes the results of the analysis less sensitive to semantic preserving rearrangements of statements in the program.

Like [91, 92], our role analysis performs non-local inference of program properties including the synthesis of loop invariants. The difference is that [91, 92] focus on linear constraints between integers and handle recursive data structures conservatively, whereas we do not handle integer arithmetic but have a more precise representation of the heap that captures the constraints between objects participating in multiple data structures.

7.2 Roles in Object-Oriented Programming

It is widely recognized that conventional mechanisms in object-oriented programming languages do not provide sufficient control over object aliasing. As a result, it is not possible to prevent *representation exposure* [22] for linked data structures. As some previous systems, our roles can be used to avoid representation exposure, even though this is not the only purpose of roles.

Islands [46] were designed to help reasoning about object-oriented programs. An island is a set of objects dominated by a *bridge* object in the graph representing the heap. To keep track of aliasing, [46] introduces *unique* and *free* variables with reference counts zero and one, respectively. It also defines a destructive read operation which can be used to pass free objects into procedures. Roles can also be used to enforce the invariant that an object dominates a set of objects reachable along a given set of fields by specifying slot constraints that prevent aliases from objects outside the data structure. Our slot constraints substantially generalize unique and free variables. Our role analysis uses precise shape analysis techniques, which is in sharp contrast with purely syntactic rules of [46].

Balloon types [4] is another system that supports encapsulation. It requires minimal program annotations. The encapsulation in balloon types is enforced using abstract interpretation. The analysis representation records reachability status between objects referenced by variables and relationship of these objects with *clusters* of objects. In most cases our role analysis is more precise than [4] because we track the aliasing properties of objects in recursive data structures, and not only properties of paths between objects.

Ownership types [14, 66] introduce the notion of object ownership to prevent representation exposure. In contrast to the type system [14] where the owner of an object is fixed, our role analysis allows the objects to change the data structure. Furthermore, an object in our system can be simultaneously a member of multiple data structures, and the role analysis verifies the movements of objects specified in procedure interfaces.

The object-oriented community has also become aware of the benefits of the systems where the class of an object changes over the course of the computation. Predicate classes [11] describe objects whose class depends on values of arbitrary predicates. The system [11] computes the values of predicates at run-time and does not attempt to statically infer values of these predicates, leaving to the user even the responsibility of ensuring the disjointness of predicates for incomparable classes. One of the features of predicate classes is a dynamic dispatch based on the current class of the object. In contrast, we are proposing a a selected family of heap constraints and a static role analysis that keeps track of these constraints. Our role system does not have dynamic dispatch. Instead, the declared roles of parameters define a precondition on a procedure call. This precondition changes the operations applicable for an object based on the statically computable information about the dynamic state of the object. Finally, [11] does not attempt to define the state of an object based on object's aliases, which is the central idea of our approach. Even with the great freedom gained by giving up the static checking of classes, systems like [11] cannot verify invariants expressed with our slot constraints; this would in general require adding additional instrumentation fields that track the inverse references.

Dynamic object re-classification [26] presents a system closer to the conventional class-based languages, with method invocation implemented through double dynamic dispatch. The proposal [26] does not statically analyze heap constraints. The work [93] describes a system inspired by a knowledge based reasoning system. The object re-classification in [93] is also implemented by the run-time system. Other approaches propose using design patterns to overcome the absence of language support for dynamically changing classes [33, 29, 40, 86].

The term "role" as used in object-oriented modeling and object-oriented database communities is different from our concept of roles. A role of an object in these systems does not capture object's aliasing properties and other heap constraints. In [70], role denotes the purpose of an object in a collaboration [86] or a design pattern. Our concept of roles captures the associations between objects in a pattern by specifying references that originate or terminate at that object. As in our system, the role of an object in [70] changes over time, and an objects can play multiple roles simultaneously, which corresponds to our partial roles. Our role system ensures the conformance of these design concepts with the actual implementation, improving the reliability of the application. In the database programming language Fibonacci [35, 3] each object plays multiple roles simultaneously. The interface of an object depends on the role through which the object is accessed. This is in contrast to our role system where the role is a structural property of an object. As in most other database implementations, the system [3] checks the inclusion and cardinality constraints on associations at run-time, unlike our static analysis.

7.3 Shape Analysis

The precision of our role analysis for tracking references between heap objects is closest to the precision of the shape analysis and verification techniques such as [78, 32, 54, 36]. Whereas these systems focus on analyzing a single data structure, our goal is to analyze interactions between multiple data structures. This is reflected in our choice of the properties to analyze. In particular, the slot constraints tracked by our role analysis are a natural generalization of the sharing predicate in [78] and can be used both to refine the descriptions of data structure nodes and to specify the membership of objects in multiple data structures.

Shape Types [32] is a system for ensuring that the program heap conforms to a context-free graph grammar [27, 73]. As a graph description formalism, context-free graph grammars are incomparable to roles. On the one hand, graph grammars cannot describe an approximation of sparse matrices or specify participation of objects in multiple data structures. On the other hand, the nonparametrized role system presented in this thesis does not include constraints such as "a node must have a self loop". We could express such constraints using roles parametrized by objects. The problem of temporary violations of heap invariants is circumvented in [32] by using high-level graph rewrite rules called *reactions* [30] as part of the implementation language. The model [32] does not support nested reactions on the same data structure or procedure calls from reactions. In contrast, the model of onstage and offstage nodes can be directly applied to a Java-like language, and gives more flexibility to the programmer because roles can be violated in one part of data structure while invoking a procedure on disjoint part of the same data structure. There is no support for procedure specifications in [32]. While simple procedures might be described precisely as reactions, for larger procedures it is necessary to use approximations to keep procedure summaries concise. Our system achieves this goal by using effects as nodeterministic procedure specifications that enable compositional interprocedural analysis.

Graph types and the pointer assertion logic [54, 52, 64] are heap invariant description languages based on monadic second-order logic [85, 17, 55]. In these systems, each graph type data structure must be represented as a spanning tree with additional pointer fields [64] constrained to denote exactly one target node. If a data structure is expressible in this way, the system [64] can verify strong properties about it, an example is manipulation of a threaded tree. Because of constraints on pointer fields, however, it is not possible to approximate data structures such as trees with a pointer to the last accessed leaf, skip lists, or sparse matrices. This restriction also makes it impossible to describe objects that move between data structures while being members of multiple data structures simultaneously. The moving objects cannot be made part of any backbone because their membership in data structures changes over time. The verification of programs in [64] is based on loop invariants. This makes the technique naturally modular and hence no special mechanism is needed for interprocedural analysis. Because the logic is second order, the effects of the procedure can be specified by referring to the sets of nodes affected by the procedure. The problem with this approach is the complexity of loop invariants that describe the intermediate referencing relationships. In contrast, our role analysis uses fixpoint computation to effectively infer loop invariants in the form of sets of role graphs and uses procedures as a unit of a compositional interprocedural analysis.

Like shape analysis techniques [12, 36, 77, 78], we have adopted a constraint-based approach for describing the heap. The constraint based approach allows us to handle a wider range of data structures while potentially giving up some precision.

The path matrix approaches [37, 36] have been used to implement efficient interprocedural analyses that infer one level of referencing relationships, but are not sufficiently precise to track must aliases of heap objects for programs with destructive updates of more complex data structures.

The ADDS data structure description language [49] uses declarations of unique pointers and independent data structure *dimensions* to communicate data structures invariants. Later systems [50, 45] replace these constraints with reachability axioms. None of these systems has a concept of a role which depends on aliasing of an object from other objects. These systems use sound techniques to apply the data structure invariants for parallelization and general dependence testing but do not verify that the data structure invariants are preserved by destructive updates of data structures [48].

The use of the instantiation relation in role analysis is analogous to the materialization operation of [77, 78]. The shape analysis [77, 78] uses abstract interpretation [18] to compute the invariants that the program satisfies at each program point. The values of invariants are stored as 3-valued models for the user-supplied instrumentation predicates. In contrast, our analysis representation is designed to verify a particular role programming model with onstage and offstage nodes. Role graphs use "may" interpretation of edges for offstage nodes and "must" interpretation of edges adjacent to onstage nodes. The abstraction relation is based on graph homomorphism and it is not necessarily a function, so there is no unique best abstract transformer as in the abstract interpretation frameworks. Our role analysis can thus create the summary nodes with different reachability predicates on demand, depending on the behavior of the program. Next, the possibility of having multiple role assignments with static analysis based on the instrumented semantics allows us to capture certain properties of objects that depend not only on the current state of the heap but also on the computation history. Reachability properties in our role analysis are derived from the role graph instead of being explicitly stored as instrumentation predicates. The advantage of our approach is that it naturally handles a class of reachability predicates, without requiring predicate update formulae. Our approach thus avoids the danger of a developer supplying incorrect predicate update formulae and thereby compromising the soundness of the analysis. A disadvantage of our approach is that it does not give must reachability information for paths containing several types of fields where nodes have multiple aliases from those fields. The reason why we *can* recover reachability for e.g. tree-like data structures is that the slot constraint in a role which labels a summary node guarantees the existence of the parent for each node in the path. Our role analysis handles acyclicity by using roles to store the acyclicity assumptions for nodes in recursive data structures. Acyclicity assumptions are instantiated using the the split operation. Our split operation achieves a similar goal to the focus operation of [78]. However, the generic focus algorithm of [60] cannot handle the reachability predicate which is needed for our split operation. This is because it conservatively refuses to focus on edges between two summary nodes to avoid generating an infinite number of structures. Rather than requiring definite values for reachability predicate, our role analysis splits according to reachability properties in the abstract role graph, which illustrates the flexibility of the homomorphism-based abstraction relation.

Type inference algorithms for dynamically typed functional languages [2, 10] have the ability to statically approximate the values of types in higher order languages. These systems usually work with purely functional subsets of functional languages and do not consider the issues of aliasing.

7.4 Interprocedural Analyses

A precise interprocedural analysis [72] extends the shape analysis techniques to treat activation records as dynamically allocated structures. The approach also effectively synthesizes an application-specific set of contexts. Our approach differs in that it uses a less precise but more scalable treatment of procedures. It also uses a compositional approach that analyzes each procedure once to verify that it conforms to its specification.

Interprocedural context-sensitive pointer analyses [90, 38, 13] typically compute points-to relationships by caching generated contexts and using fixpoint computation inside strongly connected components of the call graph. Because our analysis tracks more detailed information about the heap, we have chosen to make it compositional at the level of procedures. Our analysis achieves compositionality using procedure effects, which are also useful documentation for the procedure. Like [92] our interprocedural analysis can apply both may and must effects, but our contexts are general graphs with summary nodes and not trees.

The system [43] introduces an annotation language for optimizing libraries. The language describes procedure interfaces which enable optimization of programs that use matrix operations. The supplied function annotations are not verified for the conformance with procedure implementations. In contrast, our goal is to analyze linked data structures to verify heap invariants; it is therefore essential that our role analysis uses sound techniques for both effect verification and effect instantiation.

Our effects are more specific and precise than effects in [53]; as a result they are not commutative. Both verification and instantiation of our effects require specific techniques that precisely keep track of the correspondence between the initial heap of a procedure and the heap at each program point. Our effect application rules implement a form of effect masking. If there are no write effects with the NEW as a target and the source other than NEW, the role graphs in the caller will not be affected.

7.5 Program Verification

We can view our role analysis as one component of a general program verification system. The role analysis conservatively attempts to establish a specific class of heap invariants, but does not track other program properties. Verifying data structure invariants is important because the knowledge of these invariants is crucial for reasoning about the behavior of programs with dynamically allocated data structures, which is generally considered difficult. The difficulty of reasoning with dynamically allocated data structures is indicated by some existing systems that verify properties of interfaces but lack automatic verification of conformance between interface and implementation [42], and systems that give up soundness [28, 22]. Advances in reasoning about linked data structures [71, 51] might be a useful starting point for verification tools, although efficient manipulation of properties in verification tools results in different representation requirements than manual reasoning. A combination of model checking [47] and sound automatic model extraction [5] might be an appropriate implementation technique for verifying program properties, but the applicability of this approach for verifying heap invariants remains to be proven.

Chapter 8

Conclusion

We proposed two key ideas: aliasing relationships should determine, in large part, the state of each object, and the type system should use the resulting object states as its fundamental abstraction for describing procedure interfaces and object referencing relationships. We presented a role system that realizes these two key ideas, and described an analysis algorithm that can verify that the program correctly respects the constraints of this role system. The result is that programmers can use roles for a variety of purposes: to ensure the correctness of extended procedure interfaces that take the roles of parameters into account, to verify important data structure consistency properties, to express how procedures move objects between data structures, and to check that the program correctly implements correlated relationships between the states of multiple objects. We therefore expect roles to improve the reliability of the program and its transparency to developers and maintainers. By ensuring that the program conforms to the design constraints expressed in role definitions, role analysis makes design information available to the compilation framework. This enables a range of high-level program transformations such as automatic distribution, parallelization, and memory management.

Appendix A

Decidability Properties of Roles

This chapter presents some further results about properties of roles. The first section proves decidability of the satisfiability problem for roles with only field and slot constraints. The second section proves undecidability of the implication problem for roles.

A.1 Roles with Field and Slot Constraints

In this section we closely examine more closely properties of roles defined using solely field and slot constraints. We ignore identity and acyclicity constraints in this and the following section.

We show that we can use more general form of slot constraints without changing the expressive power of roles. We then show how the generalized slot constraints can entirely replace the field constraints, which means that these constraints are not strictly necessary once the full set of role definitions is given. Finally we show decidability of the satisfaction problem for a set of roles containing only slot constraints.

A.1.1 Forms of Slot Constraints

The particular form of our slot constraints introduced in Section 2.1.2 may seem somewhat arbitrary. In this section we introduce a more general form of slot constraints and show that it can be reduced to our original role constraints. This observation gives insight into the nature of slot constraints and is used in further sections.

Definition 49 *A* generalized slot constraint *for role r, denoted* $\mathsf{gslot}(r)$*, is a list* $c_1, \ldots, c_n$ *of* incoming configurations. *Each incoming configuration c_s is a list of pairs $\langle r_{s1}, f_{s1}\rangle, \ldots, \langle r_{sq_s}, f_{sq_s}\rangle \in R \times F$ where q_s is the length of c_s.*

By abuse of notation, we write $\langle r_j, f_j\rangle \in c_s$ if $\langle r_j, f_j\rangle$ is a member of the list c_s where c_s represents the incoming configuration.

In addition to the role assignment $\rho_c : \mathsf{nodes}(H_c) \to R$, we introduce an incoming configuration assignment $\nu : \mathsf{nodes}(H_c) \to \mathcal{N}$. For each node o, the incoming configuration assignment selects an incoming configuration $c_{\nu(o)}$ of the the role $\rho_c(o)$. The local consistency is then defined as follows.

Definition 50 $\mathsf{locallyConsistent}(o, H_c, \rho_c, \nu)$ *holds for generalized roles iff the following conditions are met. Let* $r = \rho_c(o)$.

1) *For every field* $f \in F$ *and* $\langle o, f, o' \rangle \in H_c$, $\rho_c(o') \in \mathsf{field}_f(r)$.
2) *Let* $\{\langle o_1, f_1 \rangle, \ldots, \langle o_k, f_k \rangle\} = \{\langle o', f \rangle \mid \langle o', f, o \rangle \in H_c\}$ *be the set of all aliases of node* o *and* $s = \nu(o)$. *Then* $k = q_s$ *and there exists a permutation* p *of the set* $\{1, \ldots, k\}$ *such that* $\langle \rho_c(o_{p_i}), f_{p_i} \rangle = \langle r_{si}, f_{si} \rangle$ *for* $1 \leq i \leq k$ *where* $\langle r_{si}, f_{si} \rangle$ *is the* i*-the element of the list in incoming configuration* c_s.

We say that the pair $\langle \rho_c, \nu \rangle$ of role assignment and incoming configuration assignment is valid for H_c iff $\mathsf{locallyConsistent}$ predicate holds for all nodes $o \in \mathsf{nodes}(H_c)$; the heap H_c is consistent if there exists a valid pair $\langle \rho_c, \nu \rangle$. A nonempty heap consistent with a given set of role definition is called a *model* for the role definitions.

A.1.2 Equivalence of Original and Generalized Slots

Our original slot constraints $\mathsf{slot}_i(r)$ for $1 \leq i \leq k$ where $k = \mathsf{slotno}(r)$ can be represented as generalized slot constraints with a list of all incoming configurations $c = \langle r_1, f_1 \rangle, \ldots, \langle r_k, f_k \rangle$ for $\langle r_i, f_i \rangle \in \mathsf{slot}_i(r)$, $1 \leq i \leq k$. This representation is a direct consequence of Definitions 50 and 2.

Conversely, given a set of role definitions with generalized slots, we can construct a set of role definitions with original slots as follows. Introduce a role r/c for each incoming configuration c of role r with generalized slot constraint. Let $\mathsf{origRoles}(r)$ denote the set of new roles r/c for all incoming configurations c of r. Define field and slot constraints for r/c as follows:

$$\mathsf{field}_f(r/c) = \bigcup\{\mathsf{origRoles}(r') \mid r' \in \mathsf{field}_f(r)\}$$

$$\mathsf{slot}_i(r/c) = \{\langle r_i/c', f_i \rangle \mid c' \text{ is an incoming configuration of } r_i\}$$

where $c = \langle r_1, f_1 \rangle, \ldots, \langle r_k, f_k \rangle$. Let role assignment ρ_c assign roles with generalized slots to objects and ν be the incoming configuration assignment such that $\mathsf{locallyConsistent}$ predicate holds for all heap objects. Define the assignment of original roles by

$$\rho'_c(o) = \rho_c(o)/\nu(o)$$

Then $\mathsf{locallyConsistent}$ predicate holds for the ρ'_c assigning original roles to objects.

We will use the generalized role constraints to establish the decidability of the satisfiability problem. We first show how to eliminate field constraints.

A.1.3 Eliminating Field Constraints

In this section we argue that the field constraints are mostly subsumed by slot constraints if the entire set of role definitions is given. The constraint $r' \notin \mathsf{field}_f(r)$ can be specified as $\langle r, f \rangle \notin \mathsf{slot}_i(r')$ for all slots i in the original slot constraints. In the generalized slot constraints this conditions is specified by making sure that $\langle r, f \rangle$ is not a member of any of the incoming configurations c of role r'. In order to allow this construction to work for `null` references, we introduce multislot declaration for null_R role by defining $\langle r, f\rangle \in \mathsf{multislots}(\mathsf{null}_R)$ iff $\mathsf{null}_R \in \mathsf{field}_f(r)$.

After this transformation, the field declarations will be satisfied whenever (generalized) slot constraints and null_R multislot constraint are satisfied. In the sequel we therefore ignore the field constraints.

A.1.4 Decidability of the Satisfiability Problem

In this section we show that is is decidable to determine if a given set of role definitions (containing only field and slot constraints) has a model. We show how to reduce this question to the solvability of an integer linear programming problem.

Assume a set of role definitions for roles $R = \{r_1, \ldots, r_n\}$. Let H_c be a concrete heap, ρ_c a role assignment and ν an incoming configuration assignment. Define the following nonnegative integer variables. For every i, where $1 \leq i \leq n$, let x_i be the number of nodes with role r_i:

$$x_i = |\{o \in \mathsf{nodes}(H_c) \mid \rho(o) = r_i\}|$$

Let y_{js} be the number of nodes with role $\rho_c(r_j)$ for which ν selects the incoming configuration c_s:

$$y_{js} = |\{o \in \mathsf{nodes}(H_c) \mid \rho(o) = r_j, \nu(o) = c_s\}|$$

We also introduce the values n_{fi} denoting the number of `null` references from objects with role r_i along the field f:

$$n_{fi} = |\{\langle o, f, \texttt{null}\rangle \in H_c \mid \rho_c(o) = r_i\}|$$

Assume that $\mathsf{locallyConsistent}$ predicate holds for all objects $o \in \mathsf{nodes}(H_c)$. By partitioning the set of objects first by roles and then by incoming configurations of each role, we conclude that the following equations hold for $1 \leq j \leq n$:

$$\sum_{s=1}^{q_j} y_{js} = x_j \tag{A.1}$$

Next, let us count for each role r_i and each field $f \in F$, the number of f-references from objects in $\rho_c^{-1}(r_i)$. We assumed that each object has the field f, so counting the source of these references yields x_i. Out of these, n_{fi} are null references, and the remaining ones fill the slots of objects with incoming configurations that contain $\langle r_i, f\rangle$. We conclude that for each $f \in F$ and $1 \leq i \leq n$ the following linear equation holds:

$$x_i = n_{fi} + \sum_{\langle r_i, f\rangle \in c_s} y_{js} \tag{A.2}$$

Finally, for all $\langle r_i, f\rangle \notin \mathsf{multislots}(\mathsf{null}_R)$, we have

$$n_{fi} = 0 \tag{A.3}$$

We call equations A.1, A.2, and A.3 the *characteristic equations* of role constraints.

We concluded that characteristic equations hold for each valid role and incoming configuration assignment. We now argue that a nontrivial solution of these equations implies the existence of a heap H_c, the role assignment ρ_c and incoming configuration assignment ν such that locallyConsistent predicate is satisfied for all objects of the heap.

Assume that there is a nontrivial solution of the characteristic equations. Construct a heap H_c with N nodes where $N = \sum_{i=1} x_i$. Partition the nodes of the heap into n classes and assign $\rho_c(o) = r_i$ for nodes in class i, such that the definition of x_i is satisfied for every i. This is possible by the choice of N. Next, partition each class $\rho_c^{-1}(r_i)$ into disjoint sets, one set for each incoming configuration, and assign $\nu(o) = c_s$ such that the definitions of y_{js} are satisfied. This is always possible because equation A.1 holds. Next, add edges to graph H_c so that slot constraints are satisfied. This can be done by a simple greedy algorithm which adds one edge at a time so that it does not violate any slot constraints. This construction is guaranteed to succeed because of equation A.2. The condition A.3 guarantees that the resulting graph null references will be present only for the fields for which they are allowed. The result is a heap H_c consistent with the role definitions.

The next theorem follows directly from the previous argument and the decidability of the integer linear programming problem.

Theorem 51 *It is decidable to determine if there exists a model for a given set of role definitions.*

In addition to showing the decidability, the preceding argument also illustrates that slot and field constraints are insensitive to graph operations that switch the source of a reference from object o_1 to object o_2, as long as $\rho_c(o_1) = \rho_c(o_2)$. This implies that certain heap properties are not expressible using slot and field constraints alone. In particular, slot constraints do not prevent cycles, which justifies introducing the acyclicity constraints into the role framework.

A.2 Undecidability of Model Inclusion

In this section we explore the decidability of the question "is the set of models of one set of role definitions S_1 included in the set of models of another set of role definitions S_2". This appears to be a more difficult problem than satisfiability of role definitions. Indeed, we proved in Section A.1.4 that the satisfiability is decidable for a restricted class of role definitions; in this section we prove that the model inclusion problem is undecidable for acyclic models.

Our role specifications are interpreted with respect to graphs which need not be trees and can even contain cycles. It can therefore be expected that strong enough properties are undecidable for such broad class of models. A common technique to prove undecidability for problems on general graphs is to consider the class of graphs called *grids*.

We define a grid as a labelled graph with edges x along the x-axis and edges y along the y axis.

Definition 52 *A grid $m \times n$ where $m, n \geq 5$ is any graph isomorphic to the graph with nodes*

$$V = \{1, \dots, m\} \times \{1, \dots, n\}$$

and edges $E = E_r \cup E_d$ where

$$E_x = \{\langle \langle i, j \rangle, x, \langle i + j, j \rangle \rangle \mid 1 \leq i \leq m - 1, 1 \leq j \leq n\}$$

$$E_y = \{\langle \langle i, j \rangle, y, \langle i, j + 1 \rangle \rangle \mid 1 \leq i \leq m, 1 \leq j \leq n - 1\}$$

The idea is to reduce the existence of a Turing machine computation history [81, 67] to the problem on graphs considered. The rules for computation history are local and thus can be expressed using slots and fields. However, it is not possible to use roles to directly express the condition that a graph is a grid. The problem is that the commutativity condition $o.x.y = o.y.x$ for grids cannot be captured using our role constraints, as the following reasoning shows.

Assume that there are role definitions which describe the class of grids. Since grids do not have any identities $\langle f, g \rangle$, we may assume that these role definitions do not contain identity declarations. Because the number of roles and incoming configurations is finite, there exists a sufficiently large grid E, a valid role assignment ρ_c and a valid incoming configuration assignment ν such that for some i, j where $2 < i < j$, all of the following conditions hold:

$$\begin{array}{rcl} \rho_c(\langle i, 2 \rangle) & = & \rho_c(\langle j, 2 \rangle) \\ \rho_c(\langle i, 3 \rangle) & = & \rho_c(\langle j, 3 \rangle) \\ \nu(\langle i, 2 \rangle) & = & \nu(\langle j, 2 \rangle) \\ \nu(\langle i, 2 \rangle) & = & \nu(\langle j, 2 \rangle) \end{array}$$

Define a new graph E' in the following way (see Figure A-1).

$$\begin{array}{rcl} E' & = & (E \setminus \{\langle \langle i, 2 \rangle, x, \langle i, 3 \rangle \rangle, \langle \langle j, 2 \rangle, x, \langle j, 3 \rangle \rangle \}) \\ & & \cup \{\langle \langle i, 2 \rangle, x, \langle j, 3 \rangle \rangle, \langle \langle j, 2 \rangle, x, \langle i, 3 \rangle \rangle \} \end{array}$$

We claim that the new graph E' also satisfies the same role and incoming configuration assignment. To see this, observe that the field and slot constraints remain satisfied because the new edges connect nodes with same roles as in E, there are no identities in role definitions, and the graph remains acyclic so acyclicity conditions cannot be violated. But E' is not isomorphic to a grid, because every isomorphism would have to be identity function on node $\langle 1, 1 \rangle$, and therefore also identity on all nodes $\langle 1, i \rangle$ for $i > 1$. Next, since y-edges in E' are the same as in E, the isomorphism would have to be identity function on all nodes, and this is not possible due to the change performed in the set of x-edges. We conclude there is no set of role definitions that captures the class of grids.

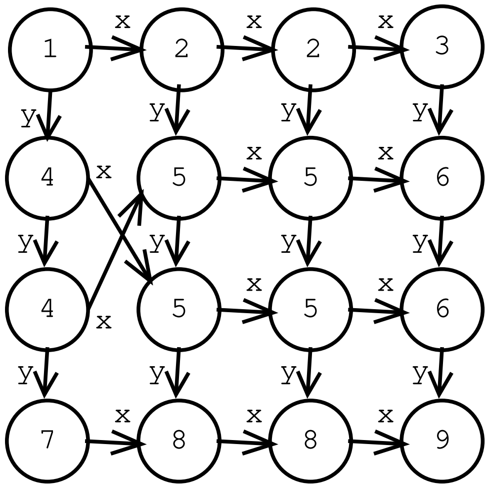


Figure A-1: A Grid after Role Preserving Modification

The idea of our undecidability construction is to use one set of role definitions S_1 to approximate the grid up to the commutativity condition $o.x.y = o.y.x$ as well as to encode the transitions of a Turing machine. We then use the another set of role definitions S_2 to express the *negation* of the commutativity condition. The models of S_1 are not included in models of S_2 if and only if there exists a model for S_1 which is not a model of S_2. Any such model will have to be a grid because it satisfies S_1 but not S_2, and the roles of S_1 will encode the accepting Turing machine computation history. Hence the question whether such a model exists will be equivalent to the existence of an accepting Turing machine computation history and the undecidability of model inclusion will follow from the undecidability of the halting problem.

Let us first consider how S_1 and S_2 define the grid used to encode the computation histories. Without the loss of generality, we restrict ourselves to models that are connected graphs. We define S_1 to be a refinement of the definition for a sparse matrix from Example 3, Figure 2-1. From properties in Section 2.3 we conclude that the connected models of E are graphs for which there exist $m, n \geq 3$ such that:

1. there is exactly one node `A1`, one node `A3`, one node `A7` and one node `A9`;

2. there are $m - 2$ nodes `A2` (by the choice of m);

3. there are $m - 2$ nodes `A8` because the acyclic lists along y establish bijection with `A2` nodes;

4. there are $n - 2$ nodes `A4` (by the choice of n);

5. there are $n - 2$ nodes `A6` because the acyclic lists along x establish bijection with `A4` nodes;

6. there are at least $\max(m-2, n-2)$ nodes `A5` (but not necessarily more than that).

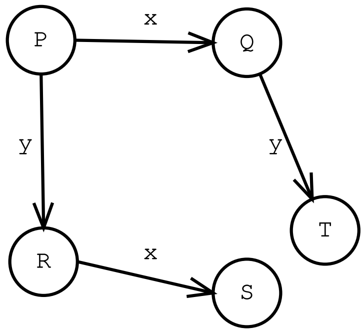


Figure A-2: Roles that Force Violation of the Commutativity Condition

The idea of role definitions S_2 is that if a graph satisfying S_1 is not a grid, then there must exist a node o such that $o.x.y \neq o.y.x$, which means that $o.x.y$ and $o.y.x$ can be assigned distinct roles. We construct S_2 to require the existence of five distinct objects o, $o.x$, $o.y$, $o.x.y$ and $o.y.x$ with with five distinct roles P, Q, R, and T (see Figure A-2). We require Q to be referenced from $P.x$, R to be referenced from $P.y$, T from $Q.y$ and S from $R.x$. In addition to these five roles, we include the roles that ensure that are assigned to the remaining nodes of a graph. We construct these roles to ensure that every model of S_2 contains an object of P role, relying on Property 12.

Finally, we explain how to encode the existence of an accepting Turing machine computation history in the set of role definitions S_1. Let M be a Turing machine and w any input. We use the fact that the computation history of M on input w can be represented as a matrix, and represent the matrix as a grid. Each row of the matrix represents configuration of the Turing machine encoded as a sequence of symbols. Because all Turing machine transitions change the tape locally, there is a finite set $W_1, \ldots, W_k$ of 3×2 tiles of symbols that characterize the matrix in the following way. We call a 3×2 window in a the matrix *acceptable* if it matches a tile. We use the fact [81] that a matrix represents a computation history of M iff

$$\text{every } 3 \times 2 \text{ window in the matrix is acceptable} \tag{A.4}$$

The condition A.4 can be split into six conditions $C^{11}, C^{12}, C^{13}, C^{21}, C^{22}, C^{23}$ where C^{ij} ensures that every 3×2 window is acceptable if it starts at (i_1, j_1) where $i_1 \equiv i \pmod 3$ and $j_1 \equiv j \pmod 3$. Let each tile W_t consist of symbols a_t^{11}, a_t^{12}, a_t^{13}, a_t^{21}, a_t^{22}, a_t^{23}.

The set of role definitions S_1 is similar to roles in Example 3 except that it splits the role `A5` into multiple roles. Each new role of S_1 is a sixtuple of positions (t_s, i_s, j_s), where $1 \leq s \leq 6$, such that $a_{t_1}^{i_1 j_1} = a_{t_2}^{i_2 j_2} = \ldots = a_{t_6}^{i_6 j_6}$. Each position (t_s, i_s, j_s) in the role sixtuple ensures that one of the conditions C^{ij} is satisfied where $s = 3(i-1) + j$, using the slot constraints. Along the x field, if $j > 1$, a role with position (t, i, j)

as k-th projection can have only aliases from roles with position $(t, i, j-1)$ as k-th projection. If $j = 1$, the aliases can be from roles with $(t', i, 3)$ as the k-th projection. Analogous slot constraints are defined for y fields.

An accepting computation history of the Turing machine M exists iff there exists a matrix where all 3×2 windows are valid which in turn holds iff there exists a grid which satisfied the constraints given by role definitions S_1. A graph which satisfies role definitions S_1 is a grid iff it does not satisfy the role definitions S_2; such graph exists iff the models of S_1 are not included in models of S_2. Hence an accepting computation history of the Turing machine M exists iff the models of S_1 are not included in the models of S_2. Since the first question is undecidable, so is the model inclusion question.

Bibliography

[1] Alexander Aiken. Introduction to set constraint-based program analysis. *Science of Computer Programming*, 35:79–111, 1999.

[2] Alexander Aiken, Edward L. Wimmers, and T. K. Lakshman. Soft typing with conditional types. In *Proc. 21st ACM POPL*, pages 163–173, New York, NY, 1994.

[3] A. Albano, R. Bergamini, G. Ghelli, and R. Orsini. An introduction to the database programming language Fibonacci. *The VLDB Journal*, 4(3), 1995.

[4] Paulo Sergio Almeida. Balloon types: Controlling sharing of state in data types. In *Proc. 11th ECOOP*, 1997.

[5] Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of C programs. In *Proc. ACM PLDI*, 2001.

[6] Barendsen and J. E. W. Smetsers. Conventional and uniqueness typing in graph rewrite systems. In *Proceedings of the 13th Conference on the Foundations of Software Technology and Theoretical Computer Science*. Springer-Verlag, 1993.

[7] Michael Benedikt, Thomas Reps, and Mooly Sagiv. A decidable logic for linked data structures. In *Proc. 8th ESOP*, 1999.

[8] Egon Börger, Erich Gräedel, and Yuri Gurevich. *The Classical Decision Problem*. Springer-Verlag, 1997.

[9] Nicolas Bourbaki. *Theory of Sets*. Paris, Hermann, 1968.

[10] R. Cartwright and M. Fagan. Soft typing. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation (PLDI)*, number 6 in 26, pages 278–292, 1991.

[11] Craig Chambers. Predicate classes. In *Proc. 7th ECOOP*, pages 268–296, 1993.

[12] David R. Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. In *Proc. ACM PLDI*, 1990.

[13] Ramkrishna Chatterjee, Barbara G. Ryder, and William Landi. Relevant context inference. In *Proc. 26th ACM POPL*, pages 133–146, 1999.

[14] David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. In *Proc. 13th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 1998.

[15] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available on: `http://www.grappa.univ-lille3.fr/tata`, 1997. release 1999.

[16] James C. Corbett. Using shape analysis to reduce finite-state models of concurrent java programs. *Software Engineering and Methodology*, 9(1):51–93, 2000.

[17] Bruno Courcelle. The expression of graph properties and graph transformations in monadic second-order logic. In *Handbook of graph grammars and computing by graph transformations, Vol. 1 : Foundations*, chapter 5. World Scientific, 1997.

[18] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. 4th ACM POPL*, pages 238–252, 1977.

[19] Karl Crary, David Walker, and Greg Morrisett. Typed memory management in a calculus of capabilities. In *Proc. 26th ACM POPL*, 1999.

[20] Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. In *Proc. ACM PLDI*, 2001.

[21] David L. Detlefs, K. Rustan M. Leino, and Greg Nelson. Wrestling with rep exposure. Technical report, DIGITAL Systems Research Center, 1998.

[22] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. Technical Report 159, COMPAQ Systems Research Center, 1998.

[23] Alain Deutsch. Interprocedural may-alias analysis for pointers: beyond k-limiting. *SIGPLAN Notices*, 29(6):230–241, 1994.

[24] Amer Diwan, Kathryn McKinley, and J. Elliot B. Moss. Type-based alias analysis. In *Proc. ACM PLDI*, 1998.

[25] Nurit Dor, Michael Rodeh, and Mooly Sagiv. Checking cleanness in linked lists. In *Proc. 7th International Static Analysis Symposium*, 2000.

[26] S. Drossopoulou, F. Damiani, M. Dezani-Ciancaglini, and P. Giannini. Fickle: Dynamic object re-classification. In *Proc. 15th ECOOP*, LNCS 2072, pages 130–149. Springer, 2001.

[27] Joost Engelfriet. Context-free graph grammars. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages. Vol. III: Beyond Words*, chapter 3, pages 125–213. Springer, 1997.

[28] David Evans. Static detection of dynamic memory errors. In *Proc. ACM PLDI*, 1996.

[29] Martin Fowler. Dealing with roles. `http://www.martinfowler.com/apsupp/roles.pdf`, July 1997.

[30] P. Fradet and D. Le Metayer. Structured gamma. *Science of Computer Programming, SCP, 31(2-3), pp. 263-289*, 1998.

[31] Pascal Fradet, Ronan Gaugne, and Daniel Le Metayer. An inference algorithm for the static verification of pointer manipulation. Technical Report 980, IRISA, 1996.

[32] Pascal Fradet and Daniel Le Métayer. Shape types. In *Proc. 24th ACM POPL*, 1997.

[33] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlisside. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Mass., 1994.

[34] Ferenc Gecseg and Magnus Steinby. Tree languages. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages. Vol. III: Beyond Words*, chapter 1. Springer, 1997.

[35] Giorgio Ghelli and Debora Palmerini. Foundations for extensible objects with roles. In *Workshop on Foundations of Object-Oriented Languages, Paris*, 1999.

[36] Rakesh Ghiya and Laurie Hendren. Is it a tree, a DAG, or a cyclic graph? In *Proc. 23rd ACM POPL*, 1996.

[37] Rakesh Ghiya and Laurie J. Hendren. Connection analysis: A practical interprocedural heap analysis for C. In *Proc. 8th Workshop on Languages and Compilers for Parallel Computing*, 1995.

[38] Rakesh Ghiya and Laurie J. Hendren. Putting pointer analysis to work. In *Proc. 25th ACM POPL*, 1998.

[39] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Sun Microsystems, Inc., 2001.

[40] Georg Gottlob, Michael Schrefl, and Brigitte Roeck. Extending object-oriented systems with roles. *ACM Transactions on Information Systems*, 14(3), 1994.

[41] Carl A. Gunter and John C. Mitchell, editors. *Theoretical Aspects of Object-Oriented Programming*. The MIT Press, Cambridge, Mass., 1994.

[42] John Guttag and James Horning. *Larch: Languages and Tools for Formal Specification*. Springer-Verlag, 1993.

[43] Samuel Z. Guyer and Calvin Lin. An annotation language for optimizing software libraries. In *Second Conference on Domain Specific Languages*, 1999.

[44] David Harel, Dexter Kozen, and Jerzy Tiuryn. *Dynamic Logic*. The MIT Press, Cambridge, Mass., 2000.

[45] Laurie J. Hendren, Joseph Hummel, and Alexandru Nicolau. A general data dependence test for dynamic, pointer-based data structures. In *Proc. ACM PLDI*, 1994.

[46] John Hogg. Islands: Aliasing protection in object-oriented languages. In *Proc. 5th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 1991.

[47] G. J. Holzmann. The model checker spin. *IEEE Trans. on Software Engineering*, 23(5):279–295, May 1997. Special issue on Formal Methods in Software Practice.

[48] Joseph Hummel. *Data Dependence Testing in the Presence of Pointers and Pointer-Based Data Structures*. PhD thesis, Dept. of Computer Science, Univ. of California at Irvine, 1998.

[49] Joseph Hummel, Laurie J. Hendren, and Alexandru Nicolau. Abstract description of pointer data structures: An approach for improving the analysis and optimization of imperative programs. *ACM Letters on Programming Languages and Systems*, 1(3), September 1993.

[50] Joseph Hummel, Laurie J. Hendren, and Alexandru Nicolau. A language for conveying the aliasing properties of dynamic, pointer-based data structures. In *Proc. 8th International Parallel Processing Symposium*, Cancun, Mexico, April 26–29 1994.

[51] Samin Ishtiaq and Peter W. O'Hearn. BI as an assertion language for mutable data structures. In *Proc. 28th ACM POPL*, 2001.

[52] Jacob L. Jensen, Michael E. Jørgensen, Nils Klarlund, and Michael I. Schwartzbach. Automatic verification of pointer programs using monadic second order logic. In *Proc. ACM PLDI*, Las Vegas, NV, 1997.

[53] Pierre Jouvelot and David K. Gifford. Algebraic reconstruction of types and effects. In *Proc. 18th ACM POPL*, 1991.

[54] Nils Klarlund and Michael I. Schwartzbach. Graph types. In *Proc. 20th ACM POPL*, Charleston, SC, 1993.

[55] Nils Klarlund and Michael I. Schwartzbach. Graphs and decidable transductions based on edge constraints. In *Proc. 19th Colloquium on Trees and Algebra in Programming*, number 787 in LNCS, 1994.

[56] Naoki Kobayashi. Quasi-linear types. In *Proc. 26th ACM POPL*, 1999.

[57] Viktor Kuncak, Patrick Lam, and Martin Rinard. A language for role specifications. In *Proceedings of the 14th Workshop on Languages and Compilers for Parallel Computing*, volume 2624 of *Lecture Notes in Computer Science, Springer*, 2001.

[58] Viktor Kuncak, Patrick Lam, and Martin Rinard. Roles are really great! Technical Report 822, Laboratory for Computer Science, Massachusetts Institute of Technology, `http://www.mit.edu/~vkuncak/papers/`, 2001.

[59] Christopher Lapkowski and Laurie J. Hendren. Extended SSA numbering: Introducing SSA properties to languages with multi-level pointers. In *Proc. 7th International Conference on Compiler Construction*. LNCS, March 1998.

[60] Tal Lev-Ami. TVLA: A framework for kleene based logic static analyses. Master's thesis, Tel-Aviv University, Israel, 2000.

[61] Harry R. Lewis and Christos H. Papadimitriou. *Elements of the Theory of Computation*. Prentice-Hall, 1981.

[62] B. Liskov and J. M. Wing. A new definition of the subtype relation. *Proc. 7th ECOOP*, 1993.

[63] Nancy Lynch and Frits Vaandrager. Forward and backward simulations – Part I: Untimed systems. *Information and Computation, 121(2)*, 1995.

[64] Anders Møller and Michael I. Schwartzbach. The Pointer Assertion Logic Engine. In *Proc. ACM PLDI*, 2001.

[65] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.

[66] James Noble, Jan Vitek, and John Potter. Flexible alias protection. In *Proc. 12th ECOOP*, 1998.

[67] Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley, Reading, Mass., 1994.

[68] John Plevyak, Vijay Karamcheti, and Andrew A. Chien. Analysis of dynamic structures for efficient parallel execution. In *Workshop on Languages and Compilers for Parallel Architectures*, 1993.

[69] William Pugh. Skip lists: A probabilistic alternative to balanced trees. In *Communications of the ACM 33(6):668–676*, 1990.

[70] Trygve Reenskaug. *Working With Objects*. Prentice Hall, 1996.

[71] John C. Reynolds. Intuitionistic reasoning about shared mutable data structure. In *Proceedings of the Symposium in Celebration of the Work of C.A.R. Hoare*, 2000.

[72] Noam Rinetzky and Mooly Sagiv. Interprocedual shape analysis for recursive programs. In *Proc. 10th International Conference on Compiler Construction*, 2001.

[73] Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformations Vol.1*. World Scientific, 1997.

[74] Radu Rugina and Martin Rinard. Design-driven compilation. In *Proc. 10th International Conference on Compiler Construction*, 2001.

[75] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modelling Language Reference Manual*. Addison-Wesley, Reading, Mass., 1999.

[76] James R. Russell, Robert E. Strom, and Daniel M. Yellin. A checkable interface language for pointer-based structures. In *Proceedings of the workshop on Interface definition languages*, 1994.

[77] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Solving shape-analysis problems in languages with destructive updating. In *Proc. 23rd ACM POPL*, 1996.

[78] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. In *Proc. 26th ACM POPL*, 1999.

[79] E. Schonberg, J. T. Schwartz, and M. Sharir. An automatic technique for selection of data representations in Setl programs. *Transactions on Programming Languages and Systems*, 3(2):126–143, 1991.

[80] Micha Sharir and Amir Pnueli. Two approaches to interprocedural data flow analysis problems. In *Program Flow Analysis: Theory and Applications*. Prentice-Hall, Inc., 1981.

[81] Michael Sipser. *Introduction to the Theory of Computation*. PWS Publishing Company, 1997.

[82] Frederick Smith, David Walker, and Greg Morrisett. Alias types. In *Proc. 9th ESOP*, Berlin, Germany, March 2000.

[83] Robert E. Strom and Daniel M. Yellin. Extending typestate checking using conditional liveness analysis. *IEEE Transactions on Software Engineering*, May 1993.

[84] Robert E. Strom and Shaula Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, January 1986.

[85] Wolfgang Thomas. Languages, automata, and logic. In *Handbook of Formal Languages Vol.3: Beyond Words*. Springer-Verlag, 1997.

[86] Michael VanHilst and David Notkin. Using role components to implement collaboration-based designs. In *Proc. 11th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 1996.

[87] Philip Wadler. Linear types can change the world! In *IFIP TC 2 Working Conference on Programming Concepts and Methods, Sea of Galilee, Israel*, 1990.

[88] David Walker and Greg Morrisett. Alias types for recursive data structures. In *Workshop on Types in Compilation*, 2000.

[89] Reinhard Wilhelm, Mooly Sagiv, and Thomas Reps. Shape analysis. In *Proc. 9th International Conference on Compiler Construction*, Berlin, Germany, 2000. Springer-Verlag.

[90] R. Wilson and M. S. Lam. Efficient context-sensitive pointer analysis for C programs. In *Proc. ACM PLDI*, June 1995.

[91] Zhichen Xu, Barton Miller, and Thomas Reps. Safety checking of machine code. In *Proc. ACM PLDI*, 2000.

[92] Zhichen Xu, Thomas Reps, and Barton Miller. Typestate checking of machine code. In *Proc. 10th ESOP*, 2001.

[93] Phillip M. Yelland. Experimental classification facilities for Smalltalk. In *Proc. 6th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 1992.